The Journal of Supercomputing, 37, 47–69, 2006

Dynamic Affinity Cluster Allocation in a Shared Disks Cluster

KYUNGOH OHN
HAENGRAE CHO
Department of Computer Engineering, Yeungnam University, Gyungsan, Gyungbuk, 712–749, Korea

Abstract. A shared disks (SD) cluster couples multiple computing nodes for high performance transaction processing, and all nodes share a common database at the disk level. In the SD cluster, a front-end router selects a node for an incoming transaction to be executed. An affinity-based routing can increase the buffer hit ratio of each node by clustering transactions referencing similar data to be executed on the same node. However, the affinity-based routing is non-adaptive to the changes of the system load. This means that a specific node would be overloaded if corresponding transactions rush into the system. In this paper, we propose a new transaction routing algorithm, named *Dynamic Affinity Cluster Allocation* (DACA). DACA can make an optimal balance between the affinity-based routing and indiscriminate sharing of load in the SD cluster. As a result, DACA can increase the buffer hit ratio and reduce the frequency of inter-node buffer invalidations while achieving the dynamic load balancing.

Keywords: cluster computing; shared disks; transaction processing; transaction routing; cache coherency; affinity clustering; load balancing.

1. Introduction

A cluster is a collection of interconnected computing nodes that presents itself as one unified computing resource. Depending on the nature of disk access, there are two primary flavors of cluster architecture designs: *shared nothing* (SN) and *shared disks* (SD) [6, 17]. In the SN cluster, each node has its own set of private disks and only the owner node can directly read and write its disks. On the other hand, the SD cluster allows each node to have direct access to all disks. The SD cluster offers several advantages compared to the SN cluster, such as dynamic load balancing and seamless integration, which make it attractive for high performance transaction processing. Furthermore, the rapidly emerging technology of storage area networks (SAN) makes SD cluster the preferred choice for reasons of higher system availability and flexible data access. Recent parallel database systems using the SD cluster include IBM DB2 Parallel Edition [5] and Oracle Real Application Cluster [16].

Each node in the SD cluster has its own buffer pool and caches database pages in the buffer. Caching may substantially reduce the number of disk I/O operations by utilizing the locality of reference. However, since a particular page may be simultaneously cached in different nodes, modification of the page in any buffer invalidates copies of that page in other nodes. This necessitates the use of a *cache coherency scheme* so that the nodes always see the most recent version of database pages [3, 4, 11].

In this paper, we propose a new transaction routing algorithm. The front-end router implements the routing algorithm to select a node for an incoming transaction to be

executed. If transactions referencing similar data are clustered to be executed on the same node (*affinity node*), then the buffer hit ratio should increase. Furthermore, the level of interference among nodes due to buffer invalidation will be reduced. This concept is referred to as an *affinity-based routing* [12, 13, 18]. However, the affinity-based routing is a purely static one and non-adaptive to the changes of the system load. It does not take the current load status of each node into account while taking the routing decisions. This is particularly problematic when the load deviation of each node is quite large. For example, if transactions referencing a specific database partition rush into the system, then the corresponding affinity node will be overloaded while other nodes are idle.

To avoid overloading individual node, we have to consider a *dynamic load balancing* as well. Unfortunately, supporting both affinity-based routing and dynamic load balancing are often contradictory goals [13]. This is because the dynamic load balancing distributes the rush transactions to multiple nodes, which may hurt the advantage of affinity-based routing. In this paper, we propose a new dynamic transaction routing algorithm, named *Dynamic Affinity Cluster Allocation* (DACA). DACA is novel in the sense that it can reduce the negative effect of dynamic load balancing by adjusting dynamically the mapping of a transaction class to the affinity node(s).

This paper is organized as follows. Section 2 summarizes the related work and Section 3 presents the algorithm of DACA in detail. We have evaluated the performance of DACA under various database workloads and system configurations. Section 4 describes the experiment model, and Section 5 analyzes the experiment results. Finally, concluding remarks appear in Section 6.

2. Related work

There are very few studies on affinity-based dynamic transaction routing in the SD cluster [12]. All of related studies select the target node of incoming transactions based on affinity when the system load is evenly balanced. However, they take different approaches when a transaction class rushes into the system. In [19], incoming transactions of the rush class are spread across *all* nodes in a round-robin fashion. On the other hand, in [7], transactions are routed to the *least loaded* node. If the load of each node except the surge node (i.e., the affinity node of the rush class) is nearly equal, a routing algorithm of [7] behaves similar to that of [19].

Both studies may achieve dynamic load balancing. However, the performance improvement could be marginal due to the following two factors [19].

- In each node except the surge node, the granules in the buffer come from two partitions: the original partition logically assigned to the node and the partition accessed by the rush class. Then the node suffers from buffer contention between the partitions. The buffer hit ratio should decrease as a result.
- Transactions of the rush class are executed concurrently at multiple nodes. If one of the transactions updates a page, it may invalidate the cached version of the page in other nodes. The frequency of inter-node buffer invalidations should increase as more nodes execute transactions of the rush class.

Recently, Oracle Real Application Cluster introduces the notion of *dynamic resource remastering* to reduce the locking overhead when affinity-based routing is adopted [16]. The basic idea is to locate the locking information of a database partition to the affinity node where the database partition is most frequently accessed. Lock operations on the database partition can be processed locally as a result. However, the dynamic resource remastering is just a mechanism to exploit the advantage of affinity-based routing, but does not imply any transaction routing algorithm.

It is worthy to compare the transaction routing with the task scheduling, which was studied extensively in distributed systems [15]. Most studies concentrated on load balancing based on task processing times alone. They did not consider the effect of affinity clustering. However, we have to support the affinity-based routing as well as load balancing. This is because transaction response times are mostly determined by database related factors like disk I/O, data contention due to concurrent access, and bookkeeping overhead of database recovery than CPU waits [2]. Furthermore, load balancing algorithms in distributed systems often migrate a task to other node during its execution or decompose a task into many concurrent subtasks [8, 9, 15]. We will not consider transaction migration in this paper. This is because most transactions execute in short duration, and thus migrating a transaction between nodes does not make sense [12]. The transaction decomposition will not be considered as well, since it requires a complex commitment protocol such as 2PC that is not adopted in most SD cluster products.

Dynamic transaction routing was also studied extensively in the SN cluster [12]. Some of the studies consider both the affinity-based routing and dynamic load balancing. However, the SN cluster limits inherently the potential for load balancing. This is because the execution node of a database operation is *statically* determined by the physical database allocation. Suppose a transaction is assigned to a lightly loaded node. Then a database operation referencing a remote database partition has to be shipped to an owner node of the partition. As a result, the load of the owner node is not significantly reduced.

The SD cluster also separates database operations into local and remote ones whether the required data item is cached in the local buffer or in the remote buffers (i.e., buffers of other nodes). However, it is important to note that the separation is determined *dynamically*. Once a node caches a data item in its local buffer, following operations on the data item can be processed locally. This offers the SD cluster much flexibility for load balancing. Therefore, it is strongly required to develop a specific transaction routing algorithm for the SD cluster.

3. Transaction routing algorithm

We propose a new affinity-based dynamic transaction routing algorithm, named *Dynamic Affinity Cluster Allocation* (DACA). The goals of DACA are two-fold.

- *Maximize the effect of load balancing*: DACA adjusts dynamically the affinity relationship between nodes and transaction classes. If a transaction class rushes, DACA allocates more affinity nodes to the class. The previous affinity relationships on the nodes are changed so that their affiliated transaction classes are mapped to other nodes. Then there is *only one* transaction class executing on the affinity nodes of the rush

Table 1. Routing parameters

Parameter	Description
#N	Number of nodes
#AC	Number of ACs (#AC $\leq$ #N)
Mem(N_p)	Memory size of a node N_p
HotSet(AC_q)	Size of database partition accessed frequently by an affinity cluster AC_q
#T(N_p)	Number of active transactions at N_p
#T(AC_q)	Number of active transactions of AC_q
$\mathcal{R}(AC_q)$	Set of nodes allocated to AC_q
$\lvert\mathcal{R}(AC_q)\rvert$	Number of nodes allocated to AC_q
$\mathcal{R}^{-1}(N_p)$	Set of ACs allocated to N_p
$\lvert\mathcal{R}^{-1}(N_p)\rvert$	Number of ACs allocated to N_p
$\bar{L}(N)$	Average load of nodes ($\Sigma_{i=1}^{\#N}$(#T(N_i)) / #N)

class. As a result, DACA can alleviate the reduction of buffer hit ratio due to load balancing. Furthermore, by allocating reasonable number of nodes to the rush class, DACA can reduce the cross buffer invalidation effect.

- *Alleviate the routing overhead*: DACA adopts the notion of *affinity cluster* (AC) [18] that includes several transaction classes with high affinity to a specific portion of a database. Then the routing algorithm can be simplified by considering the load of each AC not each transaction class.

3.1. Preliminary

Table 1 summarizes the routing parameters of DACA. We assume that #AC is equal to or smaller than #N so as to minimize the load differences among ACs [18]. We also assume that there is a single front-end router in the SD cluster. The router maintains the routing parameters and allocates incoming transactions to nodes. If the router allocates a transaction of AC_i to a node N_j, it increments both #T(AC_i) and #T(N_j). When N_j informs the commitment of the transaction to the router, both numbers are decremented. In Section 3.3, we will discuss the case of multiple routers.

The router implements a *routing function*, $\mathcal{R}$, which specifies the set of nodes allocated to each AC. If $\mathcal{R}(AC_i)$ includes several nodes, incoming transactions of AC_i are routed to the nodes in a round-robin fashion. $\mathcal{R}^{-1}$ is an inverse function of $\mathcal{R}$. Initially, $\mathcal{R}(AC_i)$ is set to $\{N_i\}$, which means that transactions of AC_i are routed to N_i. As a result, for every AC_q and N_p, $|\mathcal{R}(AC_q)|$ is set to 1 and $|\mathcal{R}^{-1}(N_p)|$ may be 0 or 1 initially. It is possible to optimize the initial setting by assigning more nodes to some AC, if we expect that transactions of the AC will occupy large portion of the system load.

DACA divides the type of overload into *AC overload* and *node overload*. The AC overload implies that transactions of an AC rush into the system and additional node has to be allocated. The node overload occurs when a node N_p is allocated to several ACs and the #T(N_p) is larger than $\bar{L}(N)$ by certain amount. In the followings, we define several terminologies to model the load status of an AC and a node.

Definition 1 (AC Overload) An affinity cluster AC_q is overloaded, if
$$\#T(AC_q) \,/\, (|\mathcal{R}(AC_q)| + 1) \geq \bar{L}(N).$$

Definition 2 (Node Overload) A node N_p is overloaded, if $|\mathcal{R}^{-1}(N_p)| > 1$ and $\#T(N_p) \geq \bar{L}(N) \times \alpha$, where α is a sensitivity factor and $1 < \alpha \leq 2$.

Definition 3 (AC Underload) An affinity cluster AC_q is underloaded, if $|\mathcal{R}(AC_q)| > 1$ and $\#T(AC_q)/(|\mathcal{R}(AC_q)| - 1) < \bar{L}(N)$.

We use the number of active transactions as a measure of the load status. The same approach was also adopted in [7]. There are many other workload descriptors like CPU utilization, average response time, and average waiting time, to measure the load status. It is also possible to combine two or more descriptors. However, as was discussed in [10], any simple workload descriptor could be used to obtain near optimal performance, and major improvement can hardly be obtained when more complex descriptors are used.

A potential problem on the number of active transactions as a workload descriptor is to ignore the effect of data contention. If transactions execute in long duration or there are hot spot data items updated by many transactions, then lock conflicts would increase. The probability of deadlock becomes high as a result. To alleviate the problem of data contention, the admission control algorithm is required. For example, in [1], if some percentage of active transactions are in blocking state due to lock conflict, the router stops admitting new transactions and even aborts active transactions. We will not consider the admission control in this paper, but the practical router has to implement both the admission control and transaction routing.

3.2. Algorithm

DACA balances the load of each node according to the load status. If AC_q is overloaded, then DACA allocates additional node to AC_q by expanding $\mathcal{R}(AC_q)$. We refer this strategy as *node expansion*. On the other hand, if there is no AC overload but a node N_p is overloaded, then DACA reallocates some AC in $\mathcal{R}^{-1}(N_p)$ to other node. We refer this strategy as *AC distribution*. Finally, if AC_q is underloaded, then DACA excludes some node allocated to AC_q by reducing $\mathcal{R}(AC_q)$. This strategy is referred as *node reduction*. Now we describe each strategy in detail.

3.2.1. Node Expansion. Suppose AC_q is overloaded and $\mathcal{R}(AC_q)$ is $\{N_q\}$. Since routing new transactions of AC_q to N_q increases the response time, the router expands $\mathcal{R}(AC_q)$ to include additional node. We select the *least loaded* node, say N_k, as a candidate. This means that $\#T(N_k)$ is the minimum among other nodes. Then $\mathcal{R}(AC_q)$ is expanded to $\{N_q, N_k\}$, and incoming transactions of AC_q are routed to either N_q or N_k in a round-robin fashion.

A complicated case occurs when N_k has already been assigned to some affinity cluster, say AC_k. In this case, we first check whether AC_k is underloaded. If that is true, a node reduction strategy is applied so that $\mathcal{R}(AC_k)$ excludes N_k, and then N_k is assigned to AC_q. We describe the node reduction strategy in Section 3.2.3. If AC_k is not underloaded, $\mathcal{R}(AC_k)$ is changed to route transactions of AC_k to other node. The next section describes how to select a new node to execute transactions of AC_k. At both cases, AC_q becomes the only affinity cluster allocated to N_k.

Example 1 Suppose an example SD cluster with four nodes (#N = 4) and four ACs (#AC = 4). Initially, $\mathcal{R}(\mathrm{AC}_i)$ is set to $\{N_i\}$ for $1 \leq i \leq 4$, and for each AC, 50 transactions are in execution, i.e., $\#\mathrm{T}(\mathrm{AC}_i) = \#\mathrm{T}(\mathrm{N}_i) = 50$ for $1 \leq i \leq 4$. If the number of incoming transactions of AC_1 increases to 150, the average load of nodes $\bar{L}(N)$ becomes $\frac{150+50+50+50}{4} = 75$. In this case, AC_1 is overloaded since $\frac{150}{1+1} = 75 \geq \bar{L}(N)$. To resolve the overload state of AC_1, the router selects the least loaded node. Since the load of every node except N_1 is equal, the router may select any node as a candidate. Suppose that N_2 is selected as a candidate. Then $\mathcal{R}(\mathrm{AC}_1)$ becomes $\{N_1, N_2\}$ as a result of node expansion. $\mathcal{R}(\mathrm{AC}_2)$ can be changed to $\{N_3\}$ or $\{N_4\}$ since the load of each node is equal. Suppose that $\mathcal{R}(\mathrm{AC}_2)$ is set to $\{N_3\}$. If the load status of each AC holds for a while, the routing information and load status of each node are as follows.

	N_1	N_2	N_3	N_4
$\mathcal{R}^{-1}$	$\{\mathrm{AC}_1\}$	$\{\mathrm{AC}_1\}$	$\{\mathrm{AC}_2, \mathrm{AC}_3\}$	$\{\mathrm{AC}_4\}$
#T	75	75	100	50

The notable features of DACA's node expansion strategy are two-fold. First, DACA tries to reduce the number of nodes allocated to the overloaded AC if the load deviation of each node is not significant. This is not true in previous routing algorithms [7, 19], where transactions of the overloaded AC are routed *immediately* across all nodes. Therefore, DACA can reduce the frequency of buffer invalidations between affinity nodes of the overloaded AC. Note that as more nodes execute transactions of an AC, the buffer invalidations occur more frequently. Frequent buffer invalidations cause lower buffer hit ratio and lots of message passing to transfer database pages between nodes [3, 18].

Next, DACA prohibits allocating both an overloaded AC and other ACs to a node. As a result, DACA can achieve high buffer hit ratio for the overloaded AC. Even though several non-overloaded ACs may be allocated to a single node, we believe that efficient handling of the overloaded AC is more important to improve the overall transaction throughput. However, if the aggregate load of several ACs allocated to a node increases, the node becomes overloaded. In this case, DACA performs the AC distribution strategy described in the next section.

3.2.2. AC Distribution. AC distribution strategy is applied when a node N_p is overloaded. Among the ACs in $\mathcal{R}^{-1}(N_p)$, suppose that $\#\mathrm{T}(\mathrm{AC}_{min})$ is the minimum and $\#\mathrm{T}(\mathrm{AC}_{max})$ is the maximum. If there is a node N_k not allocated to any AC, i.e. $\mathcal{R}^{-1}(N_k) = \{\}$, then the router updates $\mathcal{R}(\mathrm{AC}_{max})$ to $\{N_k\}$. Similarly, if N_k is allocated to an underloaded AC, the router changes $\mathcal{R}(\mathrm{AC}_{max})$ to $\{N_k\}$ after performing the node reduction strategy described in the next section.

Otherwise, the router first finds another node N_k, where N_k is not overloaded and not allocated to the overloaded AC. Then the router updates $\mathcal{R}(\mathrm{AC}_{min})$ to $\{N_k\}$. Suppose that $\mathcal{R}^{-1}(N_k)$ is currently set to $\{\mathrm{AC}_k\}$. If we route both AC_k and AC_{min} to N_k, the buffer of N_k caches database pages accessed by both ACs. Since different ACs would access disjoint portion of database, frequent buffer misses could occur if the buffer is not large enough to cache the hot sets of both ACs. To avoid frequent buffer misses, AC_{min} is routed to N_k only if $\mathrm{HotSet}(\mathrm{AC}_k) + \mathrm{HotSet}(\mathrm{AC}_{min}) < \mathrm{Mem}(N_k)$. If there are one or

more nodes satisfying the constraint, the router allocates AC_{min} to the least loaded node among them. If there is no node satisfying the constraint, AC_{min} is just allocated to the least loaded node among the affinity nodes of non-overloaded ACs.

Example 2 In Example 1, the router allocates both AC_2 and AC_3 to N_3. Now consider the case of increasing the number of active transactions of AC_2. Suppose that the sensitivity factor of α is 1.6 and #T(AC_2) increases from 50 to 84. The number of transactions in other ACs is assumed to be equal to Example 1. Then #T(N_3) increases to 134 and N_3 is overloaded since $\bar{L}(N) = \frac{75+75+134+50}{4} = 83.5$ and $134 > 83.5 \times 1.6 = 133.6$. Every node is allocated to some AC and AC_1 is not underloaded. As a result, for AC_2 and AC_3, the router updates $\mathcal{R}(AC_3)$ to N_4 since #T(AC_3) < #T(AC_2). Then the routing information and load status of each node are changed as follows.

	N_1	N_2	N_3	N_4
$\mathcal{R}^{-1}$	$\{AC_1\}$	$\{AC_1\}$	$\{AC_2\}$	$\{AC_3, AC_4\}$
#T	75	75	84	100

The frequency of AC distributions depends on the sensitivity factor α. If α is slightly larger than 1, the router would perform the AC distribution frequently since it often regards a node with over-average load as an overloaded node. This is particularly problematic if the result of AC distribution makes another node be overloaded again. By setting α with much larger value, it is possible to minimize the frequency of AC distributions. However, an extremely large value of α should result in load imbalance among nodes for a long time. DACA limits the maximum value of α to 2. This is because the router allocates the least loaded AC (AC_{min}) to the least loaded node N_k, and thus the probability of N_k being overloaded again is very low when α is set to 2.

It is good idea to adjust α dynamically according to the duration of load imbalance and the frequency of load changes. Suppose α is set to 2 at Example 2. Then the AC distribution will not happen even though #T(AC_2) increases to 84. If the load status of every AC holds for a long time, it is better to force an AC distribution by reducing α to 1.6. When the load of each AC changes frequently, we had better return to the larger value of α to avoid frequent AC distributions.

3.2.3. Node Reduction. Suppose that AC_q is underloaded and $\mathcal{R}(AC_q)$ is $\{N_q, N_k\}$. If any AC or node is overloaded, the router excludes one of the nodes from $\mathcal{R}(AC_q)$, say N_k, and uses N_k to resolve the overload state. Similar to the node expansion strategy, the node reduction strategy comes to maximize the effect of dynamic load balancing. Transactions of an AC have to be routed to the smaller number of nodes if the routing strategy does not incur any AC overload.

Example 3 After applying the AC distribution of Example 2, suppose that #T(AC_1) decreases to 50. Then both #T(N_1) and #T(N_2) are 25 and $\bar{L}(N)$ becomes $\frac{25+25+84+100}{4} = 58.5$. AC_1 becomes underloaded since $\frac{50}{2-1} = 50 < 58.5$. On the other hand, N_4 is overloaded since #T(N_4) $= 100 > 58.5 \times 1.6 = 93.6$. The router excludes N_1 or

TRANSACTION_ROUTING(AC_i)

1. $\#\mathrm{T}(\mathrm{AC}_i) = \#\mathrm{T}(\mathrm{AC}_i) + 1$
2. IF $(\#\mathrm{T}(\mathrm{AC}_i)\ /\ (|\mathcal{R}(\mathrm{AC}_i)| + 1) \geq \bar{L}(N))$ // **AC_i is overloaded**
   (a) IF $(\exists N_k, |\mathcal{R}^{-1}(N_k)| = 0)$ THEN $\mathcal{R}(\mathrm{AC}_i) = \mathcal{R}(\mathrm{AC}_i) \cup \{N_k\}$;
   (b) ELSE IF $(\exists \mathrm{AC}_q, |\mathcal{R}(\mathrm{AC}_q)| > 1$ and $\#\mathrm{T}(\mathrm{AC}_q)\ /\ (|\mathcal{R}(\mathrm{AC}_q)| - 1) < \bar{L}(N))$ // **AC_q is underloaded**
      - Select $N_k \in \mathcal{R}(\mathrm{AC}_q)$;
      - $\mathcal{R}(\mathrm{AC}_q) = \mathcal{R}(\mathrm{AC}_q) - \{N_k\}$; $\mathcal{R}(\mathrm{AC}_i) = \mathcal{R}(\mathrm{AC}_i) \cup \{N_k\}$;

   (c) ELSE select N_k, where $\#\mathrm{T}(N_k)$ is the minimum among all nodes;
      - **AC_Transfer(N_k)**;
      - $\mathcal{R}(\mathrm{AC}_i) = \mathcal{R}(\mathrm{AC}_i) \cup \{N_k\}$;
3. N_p = the next node of $\mathcal{R}(\mathrm{AC}_i)$ in a round-robin order;
4. IF $(|\mathcal{R}^{-1}(N_p)| > 1$ and $\#\mathrm{T}(N_p) \geq \bar{L}(N) \times \alpha)$ // **N_p is overloaded**
   (a) Select AC_{max}, where $\#\mathrm{T}(\mathrm{AC}_{max})$ is the maximum among all ACs in $\mathcal{R}^{-1}(N_p)$;
   (b) Select AC_{min}, where $\#\mathrm{T}(\mathrm{AC}_{min})$ is the minimum among all ACs in $\mathcal{R}^{-1}(N_p)$;
   (c) IF $(\exists N_k, |\mathcal{R}^{-1}(N_k)| = 0)$ THEN $\mathcal{R}(\mathrm{AC}_{max}) = \{N_k\}$;
   (d) ELSE IF $(\exists \mathrm{AC}_q, |\mathcal{R}(\mathrm{AC}_q)| > 1$ and $\#\mathrm{T}(\mathrm{AC}_q)\ /\ (|\mathcal{R}(\mathrm{AC}_q)| - 1) < \bar{L}(N))$ // **AC_q is underloaded**
      - Select $N_k \in \mathcal{R}(\mathrm{AC}_q)$;
      - $\mathcal{R}(\mathrm{AC}_q) = \mathcal{R}(\mathrm{AC}_q) - \{N_k\}$; $\mathcal{R}(\mathrm{AC}_{max}) = \{N_k\}$;

   (e) ELSE select a node among affinity nodes of non-overloaded ACs as follows:
      - Select N_k, where $\mathrm{HotSet}(\mathrm{AC}_{min}) + \mathrm{HotSet}(\mathcal{R}^{-1}(N_k)) \leq \mathrm{Mem}(N_k)$ and $\#\mathrm{T}(N_k)$ is the minimum;
      - IF (there exists such N_k) THEN $\mathcal{R}(\mathrm{AC}_{min}) = \{N_k\}$;
      - ELSE select N_k, where $\#\mathrm{T}(N_k)$ is the minimum; $\mathcal{R}(\mathrm{AC}_{min}) = \{N_k\}$;

   (f) If $\mathcal{R}(\mathrm{AC}_i)$ is changed as a result of AC distribution, recalculate N_p from the new $\mathcal{R}(\mathrm{AC}_i)$.
5. $\#\mathrm{T}(N_p) = \#\mathrm{T}(N_p) + 1$; **Return** N_p;

Figure 1. Transaction routing algorithm of DACA.

N_2 from $\mathcal{R}(\mathrm{AC}_1)$, and then allocates the excluded node to either AC_3 or AC_4 since $\#\mathrm{T}(\mathrm{AC}_3) = \#\mathrm{T}(\mathrm{AC}_4)$. If N_2 is excluded from $\mathcal{R}(\mathrm{AC}_1)$ and allocated to AC_3, then the routing information and load status of each node are changed as follows due to the node reduction and AC distribution strategies.

	N_1	N_2	N_3	N_4
$\mathcal{R}^{-1}$	$\{\mathrm{AC}_1\}$	$\{\mathrm{AC}_3\}$	$\{\mathrm{AC}_2\}$	$\{\mathrm{AC}_4\}$
#T	50	50	84	50

3.2.4. Algorithm. Figure 1 summarizes the transaction routing algorithm of DACA. The router performs it whenever a new transaction is incoming. The input parameter is the identifier of the transaction's AC. The return value is an identifier of a node where the transaction will be assigned.

Suppose a new transaction of AC_i is incoming. The router first checks whether AC_i is overloaded (step 2). If AC_i is overloaded, the router allocates an additional node to AC_i. The node is selected among nodes where they do not have any ACs or are allocated to an underloaded AC. If there is not such a node, the router selects the least loaded

node. At step 3, the router selects a node N_p to route the incoming transaction. If N_p is overloaded, the router reduces the load of N_p with AC distribution strategy (step 4).

The AC_Transfer(N_k) function of step 2.c has a role to transfer all ACs in $\mathcal{R}^{-1}(N_k)$ to other node. The details are similar to the step 4.e. Specifically, the router selects the least loaded node that can cache the hot sets of ACs in $\mathcal{R}^{-1}(N_k)$, and assigns the node as a new affinity node of all ACs in $\mathcal{R}^{-1}(N_k)$. If there is not such a node, the router just selects the least loaded node as a new affinity node.

When a transaction commits, the router decrements both #T(AC) and #T(N) of the transaction. Note that the AC underload could be occurred as a result. In this case, the router performs the node reduction strategy similar to the step 2.b or step 4.d.

3.3. Discussion

DACA measures the load status of each AC and node by the number of active transactions. This means that the router has to update #T(AC) and #T(N) correctly whenever a transaction inputs or commits. If there is a *single* router in the SD cluster—a usual arrangement for several commercial installations today [12]—then maintaining the load status is rather cheap since every transaction inputs via the router. The router can also optimize the commit procedure by tracing average response time of each transaction class and reducing #T(AC) and #T(N) automatically without commit messages of transactions.

If there are *multiple* routers in the SD cluster, the algorithm of Figure 1 could be modified as follows. First, initial setting of routing function $\mathcal{R}$ follows from the affinity-based static routing. $\mathcal{R}$ is replicated to every router. Then each router decides where to route the incoming transactions according to $\mathcal{R}$, and maintains its own #T(AC) and #T(N). One of the routers collects periodically the values of #T(AC) and #T(N) from all of other routers, and updates $\mathcal{R}$ according to the algorithm of Figure 1. Finally, the updated $\mathcal{R}$ is broadcast to every router. Note that reducing the period of information collection can achieve rapid adaptation to load imbalance, but requires frequent message passing between routers.

4. Simulation model

We have compared the performance of DACA with other routing algorithms under a wide variety of database workloads and system configurations using a simulation model of an SD cluster. Figure 2 shows the simulation model. It was implemented using the CSIM discrete-event simulation package [14].

We model the SD cluster consisting of a single router and a global lock manager (GLM) plus a varying number of nodes, all of which are connected via a local area network. The router model consists of a *transaction generator* and a *routing manager*. The transaction generator has a role to generate transactions, each of which is modeled as a sequence of database operations, i.e., each lock request is followed by a database access operation (a read or a write). The routing manager implements a given transaction routing algorithm.

The model of each node consists of a *buffer manager*, which manages the node buffer pool using an LRU policy, and a *resource manager*, which models CPU activity and

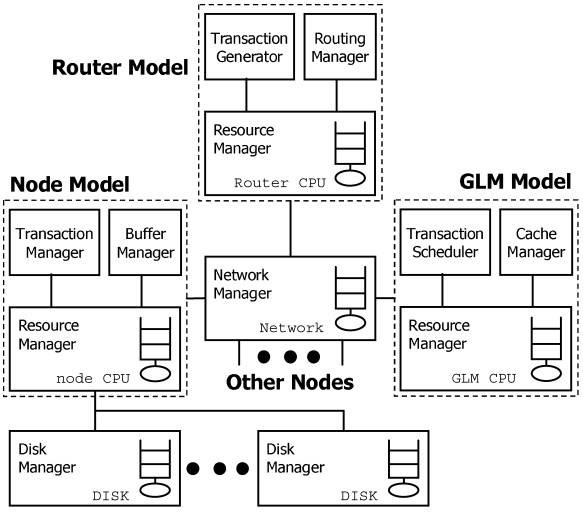

Figure 2. Simulation model of an SD cluster.

provides access to the shared disks and the network. Every node shares the disks. For each transaction, the *transaction manager* forwards lock request messages and commit messages to the GLM.

The GLM performs a concurrency control and a cache coherency control. The *transaction scheduler* supports a two-phase locking and deadlock resolution. The lock granularity is a record. The *cache manager* implements ARIES/SD algorithm [11], which is a representative cache coherency scheme in the SD cluster. We assume that the GLM's CPU performs much better than each node's CPU. The reason is to prevent the GLM from being the performance bottleneck. With the same reason, we also assume that the performance of router's CPU is identical to that of GLM's CPU.

Table 2 shows the experiment parameters for specifying the characteristics of system resources and transaction workloads. The parameter values will be used for most experiments except where otherwise noted. Many of the parameter values are adopted from [19].

The *network manager* is implemented as a FIFO server with 100 Mbps bandwidth. The CPU cost to send or receive a message via the network is modeled as a fixed per-message instruction count (*FixedMsgInst*) plus an additional per-page instruction increment(*PerMsgInst*). The number of shared disks is set to 10, and each disk has a FIFO queue of I/O requests. Disk access time is drawn from a uniform distribution between 10 milliseconds to 30 milliseconds.

We model that the database is logically partitioned into several *clusters*. Each database cluster has 10000 pages and is affiliated to a specific AC. The number of ACs is set to 1, 4, or 8. The *ACLocality* parameter determines the probability of a transaction operation accessing a page in its affiliated database cluster. The *HotProb* parameter models "80–20 rule", where 80% of the references to the affiliated database cluster go to the 20% of the database cluster (*HotSize*). We refer the 20% of the database

Table 2. Experiment parameters

	System parameters	
LCPUSpeed	Instruction rate of node CPU	500 MIPS
GCPUSpeed	Instruction rate of GLM CPU	1000 MIPS
NetBandwidth	Network bandwidth	100 Mbps
NumNode	Number of computing nodes	$1 \sim 16$
NumDisk	Number of shared disks	10
MinDiskTime	Minimum disk access time	0.01 sec
MaxDiskTime	Maximum disk access time	0.03 sec
PageSize	Size of a page	4096 bytes
RecPerPage	Number of records per page	10
ClusterSize	Number of pages in a cluster	10000
HotSize	Number of pages of hot set in a cluster	2000
DBSize	Number of clusters in database	8
BufSize	Per-node buffer size	4000
FixedMsgInst	Number of instructions per message	20000
PerMsgInst	Additional instructions per message	10000 per page
LockInst	Number of instructions per lock/unlock pair	2000
PerIOInst	Number of instructions per disk I/O	5000
PerObjInst	Number of instructions for a DB call	15000
LogIOTime	I/O time for writing a log record	0.005 sec
α	Sensitivity factor of the node overload	2.0
	Transaction parameters	
TrxSize	Number of records accessed by a transaction	10
SizeDev	Transaction size deviation	0.1
WriteOpPct	Probability of updating a record	0.3
MPL	Number of concurrent transactions	$10 \sim 640$
ACNum	Number of ACs	1, 4, 8
ACLocality	Probability of accessing local cluster	0.8
HotProb	Probability of accessing hot set	0.8

cluster as *hot set*, and the remaining part as *cold set*. The average number of records accessed by a transaction is determined by a uniform distribution between $TrxSize \pm TrxSize \times SizeDev$. The processing associated with each record, *PerObjInst*, is assumed to be 15000 instructions. Each node notifies the router whenever its transaction commits.

The performance metric is a *transaction throughput* measured as the number of transactions that commit per second. We also use an additional performance metric, *buffer hit ratio*, which gives the probability of finding the requested pages at local or remote buffers.

5. Experiment results

In this section, we compare the performance of three routing algorithms: DACA, pure affinity-based routing (PAR), and dynamic affinity-based routing that spreads transactions of rush AC to all nodes in a round-robin fashion (DRR) [19]. We first describe the performance characteristics of a single AC and then analyze the performance tradeoffs of three algorithms.

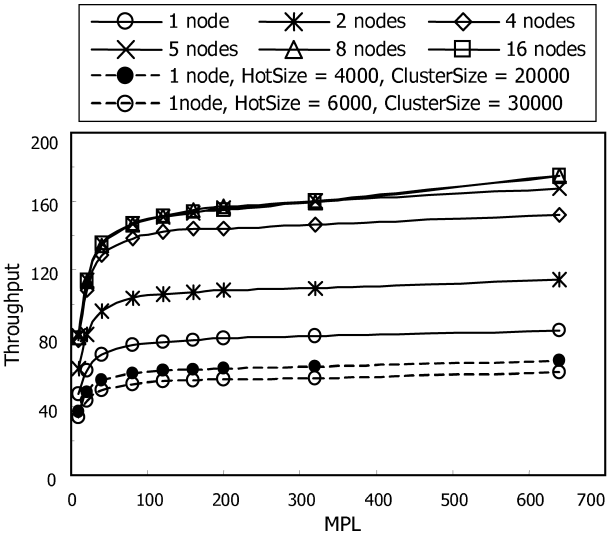

Figure 3. Transaction throughput at single AC.

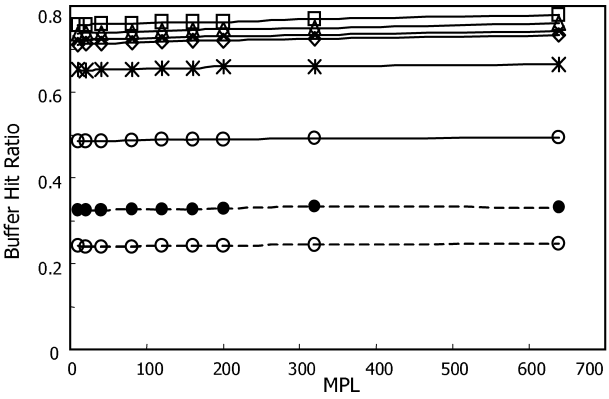

Figure 4. Buffer hit ratio at single AC.

5.1. Performance of single AC

We explore the performance of a single AC by varying *NumNode*, *HotSize*, and *ClusterSize*. Figure 3 shows the experiment results. The solid lines represent the transaction throughput when *HotSize* and *ClusterSize* are set to the values of Table 2, while *NumNode* is changed from 1 to 16. The dashed lines show the performance when *HotSize* and *ClusterSize* increase as twice and three times respectively, while *NumNode* is 1. Note that they model the situations where multiple ACs are allocated to a node.

Allocating more nodes to an AC can exploit substantial performance improvement. This is primarily due to the effect of increased buffer hit ratio as Figure 4 shows. When

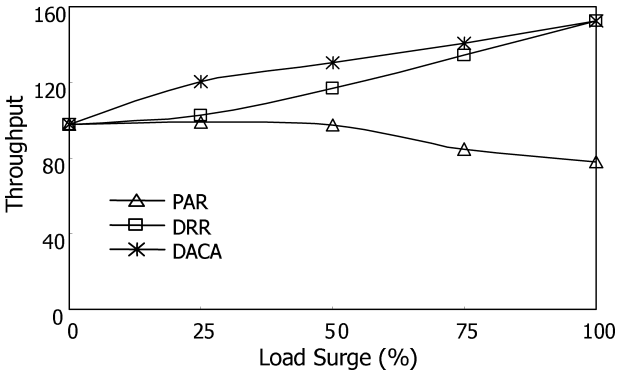

Figure 5. Transaction throughput at load surge.

NumNode is 1 and *HotSize* is 2000, the buffer hit ratio is about 0.5. Since both *ACLocality* and *HotProb* are 0.8, the probability of accessing hot set is 0.64. Then the buffer hit ratio of 0.5 implies that part of hot set may not be cached in the buffer when *NumNode* is 1. This is not surprising even though *BufSize* is the double of *HotSize*. The probability of accessing other database clusters or cold set is rather high. As *NumNode* increases, most pages in the hot set are cached at least one node's buffer, and thus the buffer hit ratio is over 0.64.

An interesting observation is that allocating large number of nodes to an AC shows only marginal performance improvement. In our experiment, allocating more than 4 nodes to an AC exhibits similar performance results. If *NumNode* is over 4, most pages in the affiliated database cluster of the AC are cached, but all of database clusters cannot be cached even *NumNode* is 16. As a result, the buffer hit ratios are nearly equal when *NumNode* is over 4. Furthermore, the potential performance improvement due to the large number of CPUs is offset by the increasing probability of buffer invalidation. This motivates our approach of DACA, where DACA allocates additional node to an overloaded AC when it is *really* beneficial.

When *HotSize* and *ClusterSize* increase to twice and three times respectively, the performance goes down due to the lower buffer hit ratio. However, their performance differences are relatively small. This is because most database accesses incur disk I/O operations and disks become the primary bottleneck.

5.2. Sudden load surge

We now compare the performance of three routing algorithms (DACA, PAR, DRR) in case of a load surge of a single AC. The extra load of the surge AC is not distributed at all in PAR, but DACA and DRR distribute the extra load to several nodes by the router.

Figure 5 compares the transaction throughput of the routing algorithms. Both *NumNode* and *ACNum* are set to 8. The value of *MPL* is 320, and thus the steady state load per each AC before the load surge is 40 transactions. The *load surge* of an AC is expressed as a fraction of its steady state load. For example, a load surge of 25% implies that the load of each non-surge AC decreases about 25% (10 transactions) and

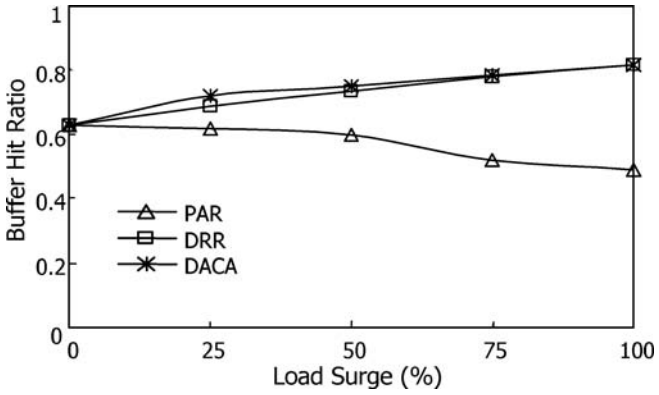

Figure 6. Buffer hit ratio at load surge.

the total sum of additional load (70 transactions) goes to the single surge AC. As a result, the load of a surge AC becomes 110 transactions, while 30 transactions are incoming per non-surge AC.

When the load surge is 0%, every algorithm performs in a similar manner. The load of each node is nearly identical in this area. As load surge increases, PAR performs worse as expected. Since PAR does not distribute the extra load of the surge AC to other nodes, PAR suffers from the lower buffer hit ratio and limited computing facility for the surge AC. Figure 6 shows that the buffer hit ratio of PAR is the lowest. As we have described in the previous experiment of a single AC, part of hot set may not be cached in the buffer when one node is allocated to an AC. This implies that the surge AC causes large number of buffer misses and disk I/O operations. When the load surge is 100%, the performance of PAR is nearly half compared to the other two algorithms. In this area, every transaction belongs to the surge AC, and the performance of PAR is equal to that of a single AC with one node in Figure 3.

Both DACA and DRR perform better as the load surge increases. When the load surge is high, most transactions belong to the surge AC and they are routed to several nodes. Allocating more nodes to the surge AC can achieve load balancing between nodes. Furthermore, the aggregate buffer space for the surge AC increases also. This is why the buffer hit ratio of both algorithms increases as the load surge increases.

DACA outperforms DRR significantly when the load surge is between 25% and 75%. At first sight, this result might be inconsistent to the buffer hit ratio of Figure 6. The buffer hit ratio of DACA is slightly higher than or nearly equal to that of DRR. Note that the buffer hit ratio of Figure 6 is the sum of a *local* buffer hit ratio and a *remote* one. For a node, the local buffer hit ratio means a probability of finding the required data item in its local buffer. A probability of finding the data item from other node's buffer is represented as the remote buffer hit ratio. Figure 7 shows each buffer hit ratio separately. Between the load surge of 25% and 75%, the local buffer hit ratio of DACA is higher than that of DRR. The reverse is true on the remote buffer hit ratio. When the remote buffer hit ratio is high, a lot of pages should be transferred between nodes. Frequent

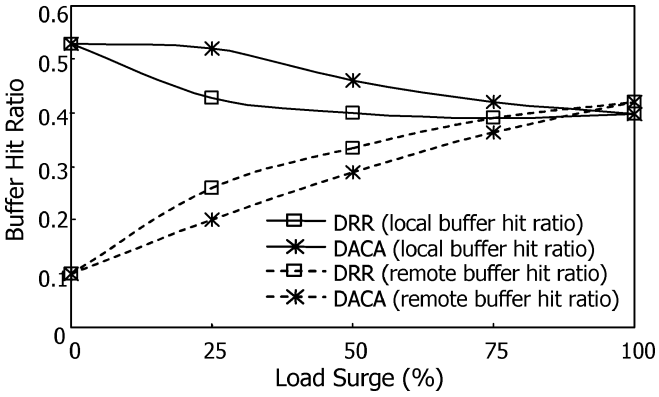

Figure 7. Local/Remote buffer hit ratio at load surge.

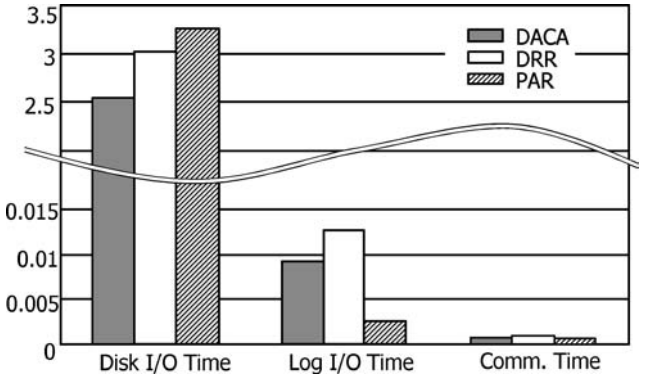

Figure 8. Average execution time per transaction commit (load surge = 25%).

page transfers limit the performance improvement due to the following reasons.

- Suppose a sender node N_s transfers a page P to a receiver node N_r. Before receiving P, N_r has to prepare a new buffer slot to store P. If the slot is already occupied by another modified page, N_r has to write the page into the disk first. As a result, the number of disk I/O operations could increase when the remote buffer hit ratio is high.
- If N_s has modified P, it has to write the log records before transferring P to ensure the write-ahead logging protocol [2, 11]. This causes additional log disk I/O operations.
- Finally, the page transfer increases the message traffic between nodes.

To evaluate the effect of page transfer, we analyzed the transaction execution time. Figure 8 shows the average disk I/O time, log I/O time, and communication time elapsed for each transaction. The communication time gives the total time for transferring messages and pages per transaction commit. The load surge was set to 25%.

Compared to PAR, DRR can reduce the disk I/O time with remote buffers. However, DRR takes more disk I/O operations than DACA due to frequent page transfers. DRR

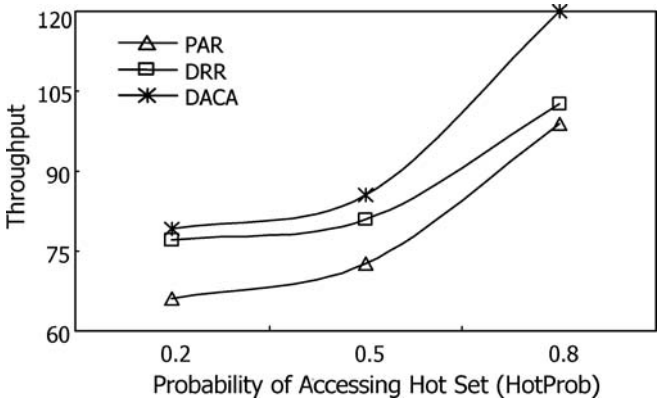

Figure 9. Transaction throughput on various *HotProb* (load surge = 25%).

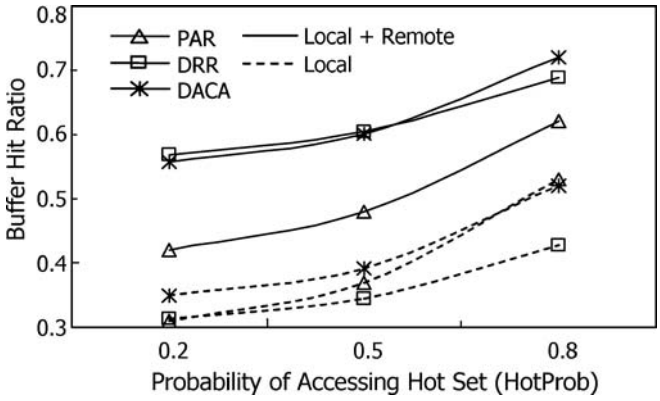

Figure 10. Buffer hit ratio on various *HotProb* (load surge = 25%).

also suffers from longer log I/O time and communication time. Note that the disk I/O time is the most important factor to determine the performance. Actually, we also performed an experiment on a Gigabit Ethernet where *NetBandwidth* is 1 Gbps. We omit the performance graph since it was nearly equal to Figure 5. Reducing the communication time had very little impact on the performance.

5.3. Performance effect of transaction access pattern

The next experiment compares the performance of routing algorithms on various transaction access patterns. Specifically, we change the probability of accessing hot set (*HotProb*). *HotProb* is one of the key parameters to determine the buffer hit ratio. Figure 9 and 10 shows the experiment results when *HotProb* is changed from 0.2 to 0.8. Both *NumNode* and *ACNum* are set to 8. The load surge is 25% and MPL is 320.

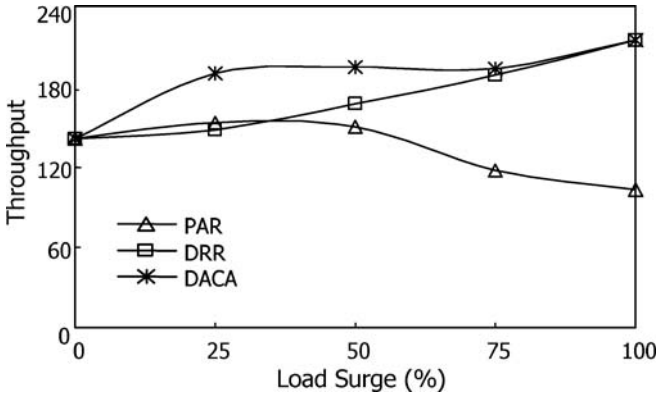

Figure 11. Transaction throughput at load surge (*NumNode* = 16).

As *HotProb* increases, every algorithm performs better. Note that *BufSize* is the double of *HotSize*. If a node has an affinity of only one AC, it can cache many hot set pages of the AC. This is why every algorithm achieves high buffer hit ratio when *HotProb* is high. Among the algorithms, the performance improvement of DRR is limited relatively. DRR distributes transactions of the surge AC to every node. When *HotProb* is high, every node accesses frequently the hot set pages of the surge AC. As a result, DRR suffers from the frequent page transfers at high *HotProb*.

PAR performs worst in every setting, especially when *HotProb* is 0.2. If *HotProb* is low, a transaction accesses more pages of cold set. Since PAR assigns each AC to only one node, a node cannot cache many pages of cold set for the surge AC. So PAR suffers from the low buffer hit ratio when *HotProb* is 0.2. Note that DACA and DRR allow several nodes to cache pages of surge AC; hence, their buffer hit ratios are rather high even at low *HotProb*.

5.4. Sensitivity to number of nodes and ACs

The number of nodes (*NumNode*) plays an important role in terms of buffer invalidation and remote buffer hit ratio. We examine the sensitivity of routing algorithms to *NumNode* by increasing *NumNode* to 16. Since *NumNode* is the double of the number of ACs (*ACNum* = 8), each AC is assigned to two nodes initially. PAR fixes the initial setting throughout the experiment. DACA and DRR change it when the load varies. Figure 11 and 12 shows the experiment results when MPL is 320.

DRR performs worst when the load surge is 25%. Since *NumNode* is large, DRR allocates more nodes to the surge AC. This increases the probability of inter-node buffer invalidation. As a result, frequent page transfers offset the benefit of load distribution. PAR performs well until the load surge is 50%. By allocating one more node, the load of the surge AC can be distributed between two nodes. However, two nodes are not enough when the load surge is over 50%. Figure 12 shows that the buffer hit ratio of PAR drops significantly when the load surge is over 50%. DACA exploits substantial

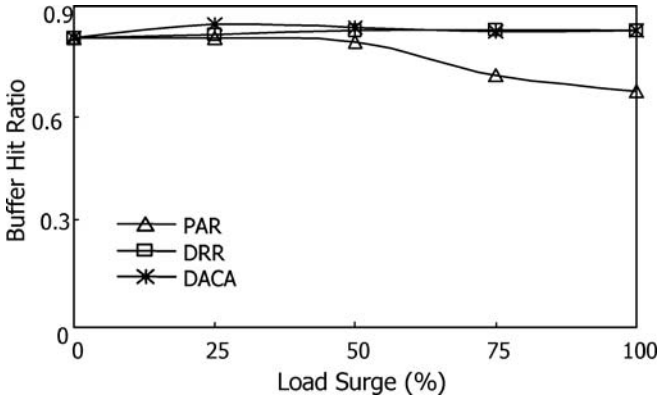

Figure 12. Buffer hit ratio at load surge (*NumNode* = 16).

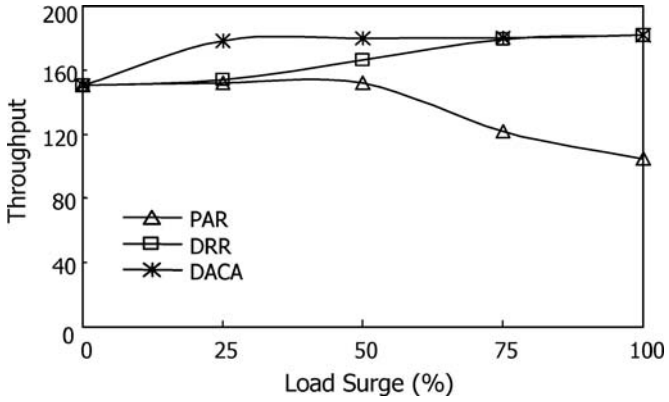

Figure 13. Transaction throughput at load surge (*ACNum* = 4, *NumNode* = 8).

performance improvement in this workload. Since there are more nodes, DACA can make more elaborate routing with dynamic affinity mapping between nodes and ACs.

We also reduce *ACNum* to 4. Figure 13 and 14 show the experiment results when *NumNode* is 8 and 4, respectively. The performance graph of Figure 13 is similar to that of Figure 11. This is because *NumNode* is the double of *ACNum* at both experiments. However, DRR now performs better than PAR when the load surge is 25%. The probability of buffer invalidation decreases due to the smaller *NumNode*. This is why DRR performs better when *NumNode* is reduced further as Figure 14 shows. When *NumNode* is 4, the performance improvement of DACA is marginal since there are limited alternatives of affinity mapping between nodes and ACs.

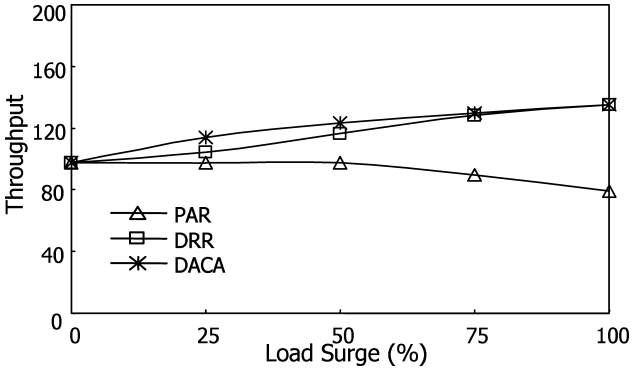

Figure 14. Transaction throughput at load surge ($ACNum = 4$, $NumNode = 4$).

5.5. *Performance effect of α*

DACA uses a sensitivity factor α to determine the frequency of AC distribution. When α is slightly larger than 1, DACA can adapt promptly to load changes. However, the frequency of AC distributions increases as a result. The reverse is true for a large value of α.

To evaluate the tradeoffs on different values of α, we perform two experiments for two versions of DACA (DACA 1.2 and DACA 2.0) where α is set to 1.2 and 2.0, respectively. Both experiments have the same initial conditions: *ACNum* and *NumNode* are set to 4, and MPL is 320. AC_1 has been overloaded for a long time with 75% load surge; hence, AC_1 is assigned to three nodes. The remaining node has executed the other ACs ($AC_2 \sim AC_4$). Now the load of AC_1 and AC_2 varies according to the experiments.

In the first experiment, the overload duration of AC_2 is much shorter than that of AC_1. Specifically, the load surge of AC_2 happens once for each interval of 500 transactions commit, and it lasts until 100 transactions commit for the interval. AC_1 is overloaded for the remaining time of the interval. Figure 15 shows the experiment result. DACA 2.0

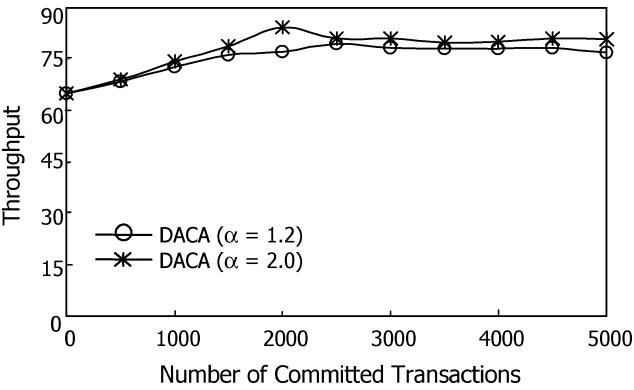

Figure 15. Performance effect of α: Short-term load change.

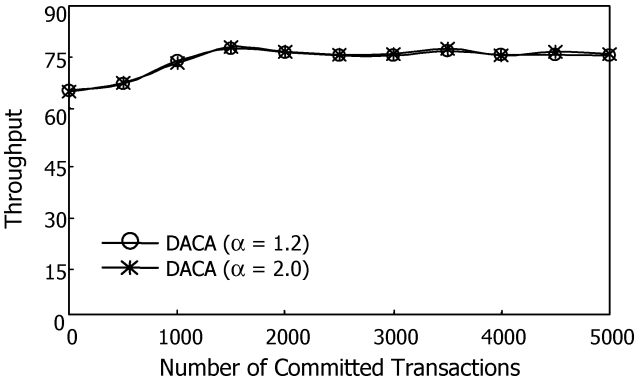

Figure 16. Performance effect of α: Long-term load change.

outperforms DACA 1.2 as expected. When α is set to 2.0, the short-term load surge of AC_2 does not cause any modification of the routing function. On the other hand, DACA 1.2 performs an AC distribution for each load surge of AC_2. As a result, the affinity mapping is changed frequently and lots of page transfers should happen between old and new affinity nodes. Since the duration of the load surge is short, the overhead of frequent page transfers offset the benefit of load distribution.

In the second experiment, we increase the load of AC_2 again but keep the overload state longer. Specifically, for each interval of 500 transactions commit, AC_2 is overloaded for the first half and then AC_1 is overloaded until the end of the interval. Now DACA 2.0 also performs AC distribution but its reaction time is slower than that of DACA 1.2. Figure 16 shows the experiment result. The purpose of this experiment was to observe the negative effect of large α due to the slow reaction time. However, the performance graphs are barely distinguishable. This means that major improvement of DACA with small α can hardly be obtained as long as DACA with large α takes the same routing decision anyway.

5.6. Overhead of multiple routers

If there are multiple routers in the cluster, DACA has to replicate the routing function between routers. Figure 17 shows the performance of DACA when the number of routers is 1 and 4. *ACNum* and *NumNode* are set to 8, and MPL is 320. The performance of PAR is also shown. Note that there is only one curve of PAR since PAR is a static algorithm. We first execute 12000 transactions with 0% load surge to make the initial environment. The initial execution result is shown at the leftmost point of each curve of Figure 17. Then one of ACs is being overloaded and the load surge varies as follows. The initial load surge is 25% until the first 2000 transactions commit, and then it increases to 50% during the next 2000 transactions commit. At the next interval, the load surge decreases to 25% again. Finally, the load surge returns to 0%. In case of multiple routers, one of the routers collects periodically the load information from other routers. The collection frequency is set to 20 times per each interval. Then the router makes a new routing function and broadcasts it to other routers.

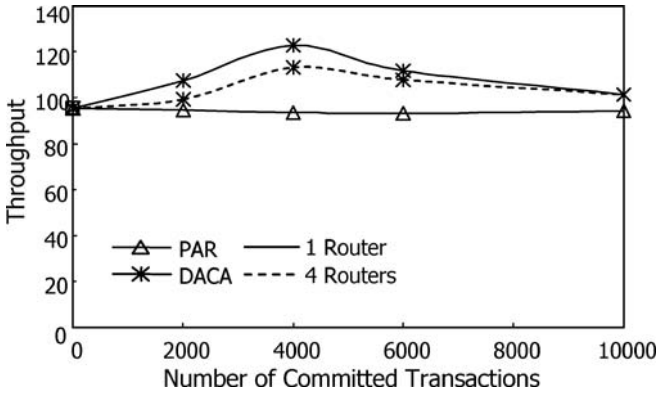

Figure 17. Overhead of multiple routers.

DACA with a single router performs better as expected. This is due to the propagation delay of multiple routers. In case of a single router, DACA can adjust the routing function promptly when the load changes. However, DACA with multiple routers have some delay to the adjustment. In our experiment, the routing functions of multiple routers become equal to that of single router after two or three collection periods for each interval. This means that enough number of nodes are not assigned to the surge AC initially at the first and second intervals. So DACA cannot distribute properly the load of surge AC. Furthermore, at the third and fourth intervals, too many nodes are assigned to the surge AC; hence, DACA with multiple routers suffer from frequent inter-node buffer invalidations and load imbalance.

It is important to note that Figure 17 shows just the overhead of multiple routers. It does not imply that a single router is always helpful. A single router has a reliability problem since the cluster has to stop when the router fails. Furthermore, multiple routers would be mandatory in some environment where there are a lot of users or the communication capability of a router is limited.

6. Concluding remarks

Clustering multiple computing nodes for database transaction processing has received considerable interest in recent years for reasons of capacity, availability and cost. Furthermore, the rapid growth of cluster hardware technologies such as storage area networks enables to develop large-scale database clusters. In this paper, we proposed a new transaction routing algorithm, named DACA, for an SD cluster. DACA is novel in the sense that it can achieve an optimal balance between the affinity-based routing and dynamic load balancing.

We have compared the performance of DACA with other routing algorithms under a wide variety of database workloads and system configurations using an SD cluster simulation model. The basic results obtained from the experiments can be summarized as follows.

- DACA outperforms the pure affinity-based routing algorithm significantly when transaction workload is changed dynamically due to sudden load surge.
- DACA also outperforms other affinity-based dynamic transaction routing algorithms by judicious node expansion strategy. The previous algorithms suffer from the frequent buffer invalidations and thus take a low local buffer hit ratio. On the other hand, DACA can achieve a high local buffer hit ratio with reduced buffer invalidations by limiting the number of nodes allocated to an overloaded AC if the load deviation of each node is not significant.
- The performance improvement of DACA is especially distinguished when there are a lot of nodes. This result is very encouraging by considering the ever increasing scalability of the SD cluster.

Acknowledgements

This research was supported by the MIC (Ministry of Information and Communication), Korea, under the University ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment).

References

1. M. Carey, S. Krishnamurthi, and M. Livny. Load control for locking: the 'half-and-half' approach. *Proc. 9th ACM PODS*, 72–84, (1990).
2. H. Cho. Database recovery using incomplete page versions in a multisystem data sharing environment. *Information Processing Letters*, 83(1):49–55, (2002).
3. H. Cho and J. Park. Maintaining cache coherency in a multisystem data sharing environment. *J. Syst. Architecture*, 45(4):285–303, (1998).
4. A. Dan and P. Yu. Performance analysis of buffer coherency policies in a multisystem data sharing environment. *IEEE Trans. Parallel and Distributed Systems*, 4(3):(1993), 289–305.
5. *DB2 Universal Database for OS/390 and z/OS—Data Sharing: Planning and Administration*, IBM SC26-9935–01, (2001).
6. D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Comm. ACM*, 35(6):85–98, (1992).
7. S. Haldar and D. K. Subramanian. An affinity-based dynamic load balancing protocol for distributed transaction processing systems. *Performance Evaluation*, 17(1):53–71, (1993).
8. B. Hamidzadeh, L. Y. Kit, D. J. Lilja. Dynamic task scheduling using online optimization. *IEEE Trans. Parallel and Distributed Syst.* 11(11):1151–1163, (2000).
9. V. Kanitkar and A. Delis. Site selection for real-time client request handling. In *Proc. 19th Int. Conf. on Distributed Comp. Syst.*, 298–305, (1999).
10. T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Trans. Software Eng.*, 17(7):725–730, (1991).
11. C. Mohan and I. Narang. Recovery and coherency control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proc. 17th Int. Conf. on VLDB*, 193–207, (1991).
12. C. N. Nikolaou, M. Marazakis, and G. Georgiannakis. Transaction routing for distributed OLTP systems: survey and recent results. *Information Sciences*, 97(1–2):45–82, (1997).
13. E. Rahm. A framework for workload allocation in distributed transaction processing systems. *J. Syst. Software*, 18(3):171–190, (1992).
14. H. Schwetmann. *User's Guide of CSIM18 Simulation Engine.* Mesquite Software, Inc., (1996).
15. B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, (1995).

16. M. Vallath. *Oracle Real Application Clusters*. Elsevier Digital Press, (2004).
17. M. Yousif, Shared-storage clusters, *Cluster Comp.*, 2(4):249–257, (1999).
18. P. Yu and A. Dan. Performance analysis of affinity clustering on transaction processing coupling architecture. *IEEE Trans. Knowledge and Data Eng.*, 6(5):764–786, (1994).
19. P. Yu and A. Dan. Performance evaluation of transaction processing coupling architectures for handling system dynamics. *IEEE Trans. Parallel and Distributed Syst.*, 5(2):139–153, (1994).