Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

OPTIMISTIC PARALLEL SIMULATION OF A LARGE-SCALE VIEW STORAGE SYSTEM

Garrett Yaun
Christopher D. Carothers
Sibel Adali
David Spooner

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A

ABSTRACT

In this paper we present the design and implementation of a complex view storage system model that is suitable for execution on a optimistic parallel simulation engine. What is unique over other optimistic systems is that reverse computation as opposed to state-saving is used to support the rollback mechanism. In this model, a hierarchy of view storage servers are connected to an array of client-side local disks. The term *view* refers to the output or result of a query made on the part of an application that is executing on a client machine. These queries can to be arbitrarily complex and done using SQL. In our performance study of this model, we find that speedups range from 1.5 to over 5 on 4 processors. Super-linear speedups are attributed to a slow memory subsystem and the increased availability of level-1 and level-2 cache when moving to a larger number of processors.

1 INTRODUCTION

CAVES is a configurable applications *view storage system*. This system in practice is designed to work as a middle-ware system, connecting multiple possibly distributed servers to multiple possibly distributed clients. The main purpose of any storage system is to reduce the overall turnaround time between an application and a data provider. To achieve this, storage systems make use of the possible locality between different data requests and try to optimize the overall performance by storing data that is most likely to be re-requested in the near future in a fast storage medium. In this storage system, we consider the local disk of a storage management system as a fast medium compared to networked and possibly distant servers that process complex database queries. We use *view* to refer to the output of a query in any application.

Examples of applications that involve costly queries include data warehousing and data mining applications that process complex queries over large data sets, applications that use spatial aggregations and correlations over complex vector data, biological databases that process highly complex sequence comparison algorithms, and large data sets obtained from scientific experiments. In such applications, the time to process and transmit a single query may be rather high even in the presence of indices and the size of the output may be very large. The cost of producing such a data set might be much larger than the cost of writing and reading back the data set from the local disk. In regular storage replacement algorithms, the goal of the system is to optimize the total number of hits. This assumes that the cost of producing each item in the storage is uniform. However, this is not the case for complex queries that are transmitted over a network which optimizes the overall savings in time. As a result, the replacement policies for an application may vary greatly based on its workload.

The CAVES system is designed as a general purpose storage management system that can be stored either at a middle-ware server or at a client machine. Its purpose is to store views from multiple applications and re-use them whenever possible. This system is fully programmable, making it possible for applications programmers to specify various storage management protocols that are best suited for a given application. CAVES can also work as a fast policy prototyping framework. Here, it constantly monitors the general system behavior, collects statistics and measures the effectiveness of the predefined storage management rules for a given instance. This is accomplished with the help of a simulation model of the actual system that can be run periodically to perform specific system optimizations. It is therefore crucial that the simulation model run as fast as possible.

The primary focus of this article is on the design and implementation of the CAVES simulation model for high-performance execution. In particular, this discrete-event model is *reversible* and is well suited for highly efficient execution on a optimistic parallel simulation engine. By *reversible*, we mean a simulation model that uses a reverse computation approach to achieve state-saving (Carothers, Perumalla and Fujimoto 1999(a), Carothers, Perumalla and Fujimoto 1999(b), Carothers, Perumalla and Fujimoto 1999(c), Carothers, Bauer and Pearce 2000). To the best of our knowledge, this is the first *reversible* simulation model of a storage management system.

We begin our presentation with an overview of our reversible, optimistic simulation engine, ROSS in Section 2. In Section 3, we then describe the CAVES model and how it was realized atop the ROSS engine. It is here we present how the CAVES model was made *reversible*. In Section 4, our performance results are presented. The cause and effect relationships are established between CAVES model parameters and ROSS performance. Related work is presented in Section 5 followed by our conclusions in Section 6.

2 OPTIMISTIC SIMULATION

In optimistic simulation systems (Jefferson 1985), the most common technique for realizing rollback is *state-saving*. In this technique, the original value of the state is saved before it is modified by the event computation. Upon rollback, the state is restored by copying back the saved value. An alternative technique for realizing rollback is *reverse computation* (Carothers, Perumalla and Fujimoto 1999(a), Carothers, Perumalla and Fujimoto 1999(b), Carothers, Perumalla and Fujimoto 1999(c), Carothers, Bauer and Pearce 2000). In this technique, rollback is realized by performing the inverse of the individual operations that are executed in the event computation. The system guarantees that the inverse operations recreate the application's state to the same value as before the computation.

The key property that reverse computation exploits is that a majority of the operations that modify the state variables are "constructive" in nature. That is, the undo operation for such operations requires no history. Only the most current values of the variables are required to undo the operation. For example, operators such as $++$, $--$, $+=$, $-=$, $*=$ and $/=$ belong to this category. Note, that the $*=$ and $/=$ operators require special treatment in the case of multiply or divide by zero, and overflow/underflow conditions. More complex operations such as *circular shift* (*swap* being a special case), and certain classes of random number generation also belong here.

Operations of the form $a = b$, modulo and bitwise computations that result in the loss of data are termed to be *destructive*. Typically these operations can only be restored using conventional state-saving techniques. However, we observe that many of these destructive operations are a consequence of the arrival of data contained within the event being processed. For example, the arrival of a new view in our CAVES model would change the state of the view cache by causing the eviction of a currently cached view. To solve this problem, one can simply *swap* the data in the event with the state of the logical process (LP). In this example, it is the view in the view cache that would be swapped with the view contained within the event message denoting the arrival of a new view. When the event is rolled back, the event data and LP data is just re-swapped to effect the undo operation. This solution works well as long as there is a one-to-one mapping between message data and the amount of LP state being modified. However, as we observed in our CAVES model, this is not always the case. It may come to pass that the insertion of a new view may cause the eviction of many views because of its size. Unlike typical microprocessor caches, view caches may cache items which differ in size. As we will see in the next section, this greatly complicates the reversibility of the simulation model.

It has been demonstrated that the reverse computation approach has insignificant forward computation overheads and low state memory requirements in fine-grain models. The parallel simulation performance of reverse computation has been observed to achieve better caching effects, with as much as six-fold speedup in several model configurations when compared to copy state-saving, periodic state-saving and incremental state-saving (Carothers, Perumalla and Fujimoto 1999(b)).

The CAVES model was implemented on top of ROSS, which is *Rensselaer's Optimistic Simulation System*. This optimistic simulation engine employs reverse computation to implement rollback functionality. For more detailed information on ROSS we refer to (Carothers, Bauer and Pearce 2000).

3 CAVES MODEL

The CAVES model is based on three object types: client, middle server, and the database server. The client cache server is attached to the client applications and processes requests from those applications as shown in Figure 1. When a request occurs the client cache server increments its Requests variable. Then it searches its cache for the view. If the view is found the view's priority and position in the cache get updated. The cache is sorted from lowest to the highest priority. When the priority of the view changes the view needs to be moved to the right location in the cache. Also the client increments the Hits variable and updates the Time and TimeWithOut variables. The Time variable is the total time it takes for a request to get answered with the

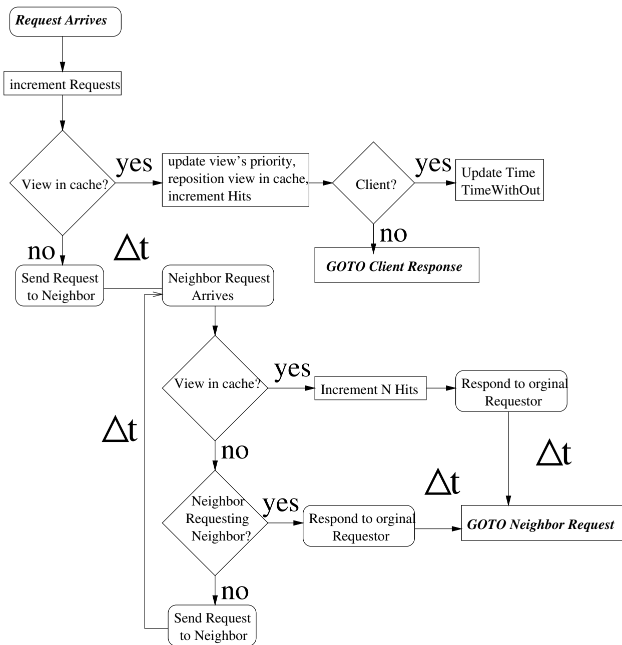

Figure 1: Flow Chart for Request Arrival and Neighbors Request

caches. The TimeWithOut is the time it would have taken without the caches.

If the view is not in the cache, the client sends a request for the view to its neighbor shown in Figure 1. The neighbor searches its cache and if the view is in its cache it sends a response back and increments the NclientHits variable. The client that receives the response updates the Time and TimeWithOut variables (Figure 2) and then it frees up room in its cache if needed. Once there is enough free space the view is inserted (Figure 4). When the client frees up room in its cache it removes views and sends those views to the middle server cache (Figure 4). The middle server checks if the view is in its cache and if so it increments the Already variable. If not then the view is inserted into its cache.

If the neighbor does not have the view it sends a request to its neighbor. The request continues to go from neighbor to neighbor until a response gets sent back or the request propagates back to the original initiator of the request. If the request gets back to the initiating client, the client then sends a request up to the middle server.

The middle server receives the request from the client cache server (Figure 1). The middle server increments its Request variable and searches its cache for the requested view. If it has the view it will respond to the client and update the view's priority and position in the cache. The middle server also updates its hits variable.

If the middle server does not have the view it will request the view from its neighbor. If the neighbor has the view it will respond and it will update the NmidHits variables. The middle server that receives the response

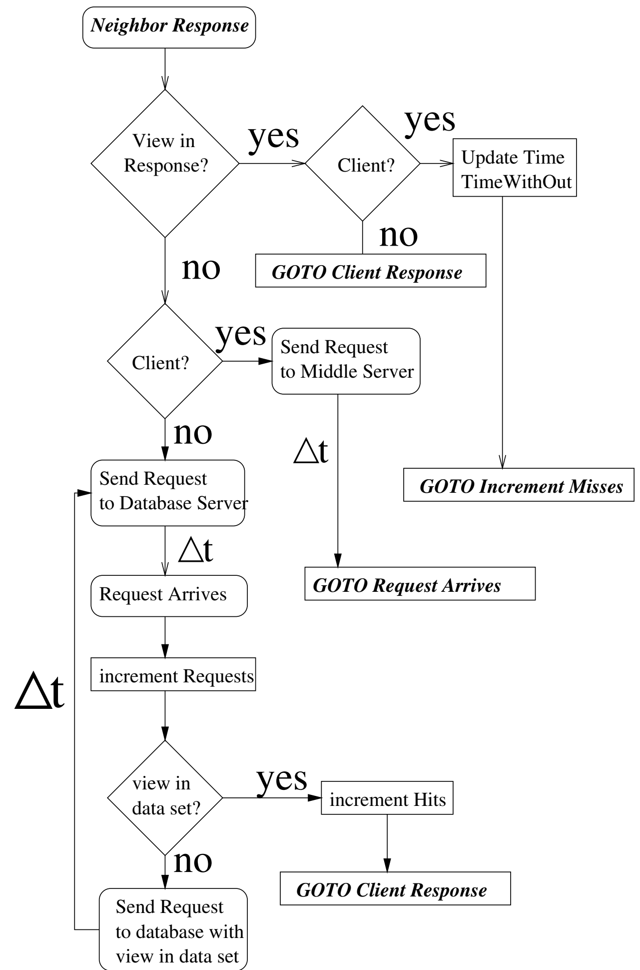

Figure 2: Flow Chart for Neighbor Response and Database Request

will then send the view back to the initiating client. If the neighbor does not have the view it will send a request to its neighbor. The requests from neighbor to neighbor will continue like they did for the client. If the request gets back around to the initiating Middle Server, it will send a request up to the Database Server.

The database server receives a request from the middle server and updates its Request variable (Figure 2). It then searches for the view in its data set. If the search finds the view (i.e., hit), it responds to the client with the view and updates the Hits variable. If the search fails, the database server requests the view from the database server with the view. That database server updates its Request variable and its Hits variable. It then responds to the client with the view.

When the client gets the response it updates the Time and TimeWithOut variables (Figure 3) and inserts the view into the cache if the view is not currently cached (Figure 4).

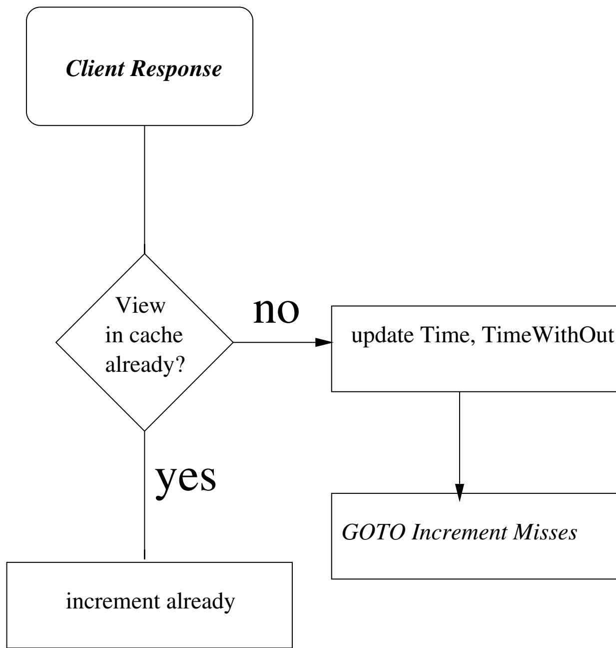

Figure 3: Flow Chart for Client Response

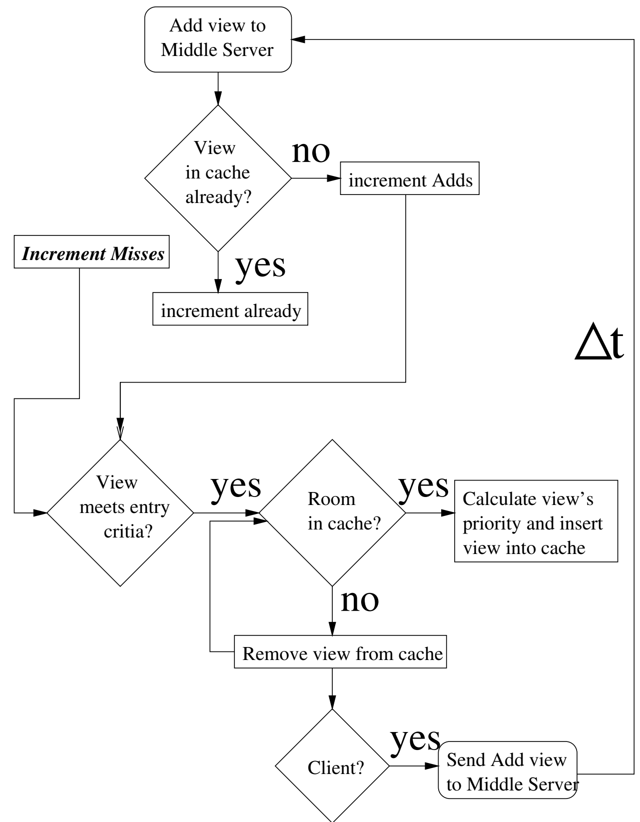

Figure 4: Flow Chart for Add View

3.1 Implementation

A natural way to map CAVES would be to make each cache server, clients and middles, along with each Database server into LPs. All the requests and responses are mapped to timestamped events that are passed from LP to LP.

Having each cache server realized as a single LP allows caching and cache management to happen sequentially. This leads to some challenges for the reverse computation because the computational complexity of these events is high. Dividing the cache server into other LPs such as a cache management LP, disk LP, and RAM LP was ruled out because the cache management LP design was to have a list of all views in the cache and their locations. Since the cache management LP has all this information it was redundant to have the disk and RAM.

Each database server and all the cache servers under it are mapped to a single KP. This was for performance reasons. With all the neighbor requests from both the client cache servers and the middle cache servers the system performance was much better when all these requests are kept within one KP. We experimented with mapping KPs to the middle cache servers. The results of this experiment show that the performance was weak. This was due to the fact that middle cache servers where sending remote neighbor requests. These requests lead to a greater number of rollbacks on average.

When the simulation starts, a global pool of views is created. Each view is given a number or id, a size, a computational complexity, and a filter factor. All variables have a range of values that are determined by the application parameters. Then by using a normal distribution, exact view values are selected based on a standard deviation and mean.

In addition, a neighbor and server table is created. These tables are used by an LP to calculate where to schedule events. In the cache server's initialization, it is allocated with a lookup table for views in the cache along with RAM and disk caches. Each cache consists of a size available parameter and two linked lists. The nodes of the linked lists are cache blocks that contain a view number, a Hits variable and a priority. There is one cache block for each view in the cache. All the allocated cache blocks are put on the free list and are later moved to the used list upon a view's entrance into the cache. A database is initialized with a range of views that it contains.

Upon the client cache servers initialization, it generates a given number of view requests uniformly from the pool of views. These view request events are scheduled into the future and will be processed by the client cache server requests method. In this method the view requested is looked up in the table to see if it is in either cache (RAM or disk). If so then the cache block, which contains the view number, Hits variable is incremented and the priority is recalculated. The view is then removed from the used list and reinserted based on its new priority. The used link list is kept with the lowest priority at the head. If it is a

miss, a request is issued to the client's neighbor. At the end of this method a new view request would be self generated and scheduled into the future for the current client LP.

When the response is received, time is increased by the amount of time it took to receive the response. At each cache server and database that processed the request, time is incremented. The time will be at least the view size divided by the network speed plus the view size divided by the disk speed.

Then the view in the response gets put into the caches. There is a write through policy implemented, so the view gets inserted to both caches. Before the view is put into the cache the size available parameter is checked to see if the view fits. If so, a cache block from the free list is removed and the view number is set. The priority of the cache block is calculated and the view is inserted into the cache. If there is not enough room, views are removed from the cache used list and put on the free list until there is enough free space. Each view that is removed generates an Add event that is sent to the middle server. A cache block is then removed from the free list, the view number is set, priority is calculated, and the view is inserted into the cache.

The above describes sequential execution of the model. To enable parallel execution, however, *reversible execution* of the model must be supported. In particular, the freeing of necessary space in the view cache in a reversible way was a problem that had to be solved. The problem stemmed from the fact that the system could remove multiple views from the cache to free up enough space for an incoming view that was significantly larger than any cached single view.

Our solution was to utilize a free-list of views. This list structure allows the view cache to be made reversible for single inserts and multiple deletes. To illustrate this functionality, consider the following example. Suppose and event is scheduled which adds view number 8 to the cache. In order to fit this view, 5 other views (1,2,3,4,5) must be removed from the cache. These 5 cache blocks are put on the free list (5,4,3,2,1). The head of the free list (5) is removed to put view 8 in. Prior to modifying the view number of the cache block, the state of that cache block is stored in the current event (i.e., swapped). We then insert 8 into the cache. Another event, which adds view 9, comes in and there is enough room in the cache to fit it. A cache block is removed for the free list (4) and the cache block's previous view information is swapped with the event data. View 9 is then put in the cache block and the cache block is inserted into the cache. The next event that comes in causes a rollback of the last two events. First we remove view 9 from the cache and set the cache block back to it's old state (4) by re-swapping the data. The cache block is then put back on the free list. We then remove view 8 from the cache and again we set the cache block back to its old state, along with putting it on the free list (5). Five views were removed from the cache to fit view 8, these views are then removed from the head of the free list and put on the head of the cache (1,2,3,4,5).

The middle cache server acts just like the client cache server in the way its cache is configured and managed. However the middle cache server never initiates a view request or inserts a neighbor's response in its cache. The middle cache only adds view that are sent up from the clients.

The database upon receiving a request looks for the view and either sends a response back to the client or redirects the request to the right database server. The database server uses the view size, filter factor, and computational complexity, along with the disk speed and network speed to calculate the time it will take to compute and transfer the view. A response event is scheduled for that time into the future to the requesting client.

3.2 Model Parameters

The CAVES model has the following application parameters with the respective configuration values used in the experiments described below: *(i) number of global views (400 and 2400), (ii) size of the clients RAM cache (2 MB and 4 MB), (iii) size of the clients disk cache (4 MB and 16 MB), (iv) request arrival time (10 seconds and 40 seconds), (v) mean view size and standard deviation (250 KB, stddev 100 KB), (vi) mean view computational complexity and standard deviation (4 with stddev of 2), (vii) mean view filter factor and standard deviation (150 with stddev of 60), (viii) disk speed of the servers (ix) network speed (100 KB/second), (x) weights of the priority functions (permutation of 0.2 and 0.6 for* $w1$*, * $w2$ *and* $w3$*), (xi) number of clients per middle server (4, 8 and 16), (xii) number of middle servers per database (4 and 8), (xiii) number of databases (4 and 8)* Each of these parameters is discussed below.

Number of global views is the number of different views that can be requested. The RAM and disk size parameters determine the amount of available space for the storing of views in the caches. The request arrival time was used to set how frequently view request would occur at the client. The view size, computational complexity and filter factor parameter are used to create the global view pool. The filter factor is the ratio of the amount of data to search through to find the view, over the view's size. The computational complexity is the difficulty level of computing the view. All of these factors go into the amount of time to receive a requested view. These factors along with the disk speed are used in the calculations of one of the priority functions.

Each priority function is given a weight. The weights of the priority functions are used to give precedence to the functions above. Currently, there are three functions, $f1$, $f2$ and $f3$, with weights $w1$, $w2$ and $w3$ respectively that

are used to calculate a view priority. Each function is given a weight. By changing the weights, the optimum combination of weights for a given workload will be utilized. The number of clients per middle server, the number of middle servers per database and the number of databases make up the number of LPs in the system. By changing the weights we hope to find the optimum combination of weights for a given workload to lead to the maximum CAVES speed up. This, however, is not in the scope of this paper. The priority is used to keep the cache sorted with the lowest priority at the head.

This set of parameter configurations resulted in 576 experiments being run for each processor. The results below focus on the 4 processor results as compared with the uniprocessor results.

4 PERFORMANCE STUDY

A number of performance metrics are used to compare and contrast the cause and effect relationships of various model parameters to ROSS performance metrics. *Event rate* is defined to be the total number of events processed less any rolled back events divided by the execution time. We define *event rate ratio* as the percentage of the event rate a particular configuration yields as compared to some base event rate. In this study, a model parameter's effect on ROSS' performance is determine by its average, best and worse case event rate ratio. *Speedup* is defined to be the event rate of the parallel case divided by event rate of the sequential. Because the total number of events are the same between sequential and parallel runs of the same model configuration, this definition is equivalent to using execution time.

Our computing testbed consists of a single quad processor Dell personal computer. Each processor is a 500 MHz Pentium III with 512 KB of level-2 cache. The total amount of available RAM is 1 GB. Four processors are used in every experiment. All memory is accessed via the PCI bus, which runs at 100 MHz.

4.1 Overall Speedup Results

Our experimental data in the aggregate showed a number of interesting trends. In particular, many of the runs showed super linear speedup, while others showed much weaker speedups. Varying of parameters for the CAVES system had an interesting effect on the Time warp system and is detailed below. The effect of the parameters on CAVES performance is beyond the scope of this paper because it deals with the complexities of changing cache policies which vary in response to changes in workload statistics.

Of the 576, four processor runs, 117 resulted in super linear speed up. Nine had a performance speed up above 5.0. The highest speed up was 5.30. These super linear speedups are attributed to the system's high workload, good look ahead, and minimum remote messages combined with four times the level-1 and level-2 cache space. In many cases where the sequential version was executing well outside the level-2 cache space, the parallel version would fit inside the level-2 cache of all 4 processors.

The minimum remote message and good look ahead can be attributed to the fact that the middle cache servers had larger caches and the global view pool was 400. Of the 117 runs, 111 had large middle caches and 107 had a view pool of 400. With the larger middle cache and smaller view pool the 117 averaged about three times more middle cache hits and 1.5 times more client cache hits than the system's average. The 117 had 2.6 times less roll backs and about two times less remote messages sent than the average.

The average speedup of the system was 3.56 on four processors and of the 576 runs, 382 had speed ups between 3 and 4. The lowest speedup, however, was 1.57. There are 77 runs with a speedup under three. This lack of performance is directly related to the fact that 50 of those runs had a large view pool and 63 had small middle caches. With the small middle cache and the large view pool the 77 runs had four times less middle hits and 1.26 times less client hits than the average of all the runs. The 77 runs had 2 times more roll backs (25% of events where rolled back) and 1.61 times more remote events than the average.

4.2 Effect of Client Cache Size

Recall from the previous discussion, our client side caches have two components: the first is a RAM cache that varies between 2 MB to 8 MB (base case for event-rate ratio calculation) and the second is a disk cache that varies from 4 MB and 32 MB. Also, We vary the client cache size from 2 MB RAM and 8 MB disk to 4 MB RAM and 32 MB disk and examine at the best and worst case performance. The average event-rate ratio was .95%.

In the worst case the event rate ratio for this configuration across all processor groups was 50%. This means by only varying the client side cache configuration, we observe a drop in event rate of 50%. However, we observe that this configuration did have the large view pool and the large mean request time. The large mean request time meant that view requests were coming in slower and therefore the workload of both runs would be less. The system, however, has a higher event granularity for misses and the small cache has more misses. This increased the smaller caches workload over the larger cache. In a system that is not heavily loaded it can get too optimistic which leads to processing of a lot of future events. The rollback distances will be much greater increasing the amount of time lost in recomputing events. The larger cache had 5890 rollbacks and 302409 event rollbacks. The smaller cache had 4578 and 65175 events rollback. The rollback distance of the

smaller cache is substantially smaller and therefore would lead to the better performance.

The best case event rate ratio across this configuration was 1.57. Thus, the larger cache configuration resulted in an even rate that was 1.57 times the event rate of the smaller cache configuration for the 4 processor case. This performance difference is attributed to the overall configuration being a case where the view pool size is small and thus has a high degree of locality, combined with the a small inter-arrival time between requests which increases the LP workload per unit of simulation time. Also, the smaller cache results in miss messages being sent to the middle level server's cache. Here, a cycle of messages is sent between the client LP and middle server LP, which results in a large number of rollbacks since the client and its middle server are mapped to the same KP.

4.3 Other Parameters

4.3.1 Effect of Middle Server Cache Size

By varying the middle cache size from 16 MB RAM and 128 MB disk down to 4 MB RAM and 32 MB disk (base case for event-rate ratio computation) we had similar performance to the client cache size parameter. The average event rate ratio across all configurations was 83%. The worst case event rate ratio was 43%. Once again the rollback distance was much farther in the large cache and therefore it was too optimistic as previous discussed. The view pool was 2400 (large) and the mean request time was 40 (high) which are the same factors that contributed to the worst case performance for the client cache size parameter. The best event rate ratio was 1.61. This time the view pool was 400, however the request time was 40 (large inter-arrival time). With a large inter-arrival time between requests, the simulation model has a tendency to run overly optimistic. In the case of the large middle server cache with small view pool, optimistic execution is allowed because few remote messages are scheduled to the off processor database server LP. In the case of the small cache, fewer hits occur at the middle tier which increases the number of remote messages sent to the database server LP. These remote messages increase the number of rollbacks and leads to slower simulator performance.

4.3.2 Effect of View Pool Size

Recall, the view size pool is the set of views from which the view generator may select. By decreasing the size of this pool, locality is increased. As the view size pool increases from 400 views to 2400 views (base case) we observe an average event rate ratio of .96. The high event rate ratio is 2.07 and the low was .64. The main different between the two is the mean arrival time. We observe that when the mean time is high the performance is best for a small pool case (i.e., 400 views). The opposite is true in the large pool case. The reason for this is because a small pool results in a higher hit-rate at both the client and middle server levels, which reduces the number of remote events and lowers the probability of rollback. With a large pool of views, more misses at the client and middle level server ensue. Thus, having a shorter inter-arrival time between requests slows the client and middle server LPs down while the database server responds. In the longer inter-arrival time case, there is less work per unit of simulation time and the likelihood of generating a rollback is increased.

4.3.3 Effect of Number of Databases

By varying the number of database servers between 8 and 4 (base case) we saw an average event-rate ratio of .98. The best event-rate ratio is 2.14 and the worse is .65. The main performance factor is the size of the caches. The 8 database case performed best with a large cache configuration and the 4 database case performed best with the smaller cache configuration. In the 8 database case with a large cache, few messages are sent between KPs. With a small cache, the likelihood of sending messages between KPs that will be rolled back increases. If one KP rolls back, it can cause the others to falsely roll back since KPs can introduce a stronger form of causality than actually exists between the LPs. In the 4 database case with small cache, there are only 4 KPs and thus the causality linkage between KPs is reduced when compared to that of the 8 database case.

4.3.4 Effect of Number of Middle Level Servers

By varying the number of middle servers from 8 to 4 (base case) we observe an average event-rate ratio of .90. The best event-rate ratio was 1.79 and the low was .70. The main factor effecting performance in this configuration is the size of the cache. For the 8 middle level server case the best event-rate ratio occurred with a large cache configuration. For the 4 middle level server case the best even-rate ratio occurred with a small cache configuration. Because each middle server could be searched to find a view, the number of hops a request takes is greater as the number of middle servers increase. This creates a dependency between those LPs and thus increases the rollback distance when a rollback does occur.

4.3.5 Effect of Number of Client Servers

By varying the number of client servers between 4, 8 and 16 we observe an average event-rate ratio of 1.02 across all configurations. Over all permutations of base case possibilities, case (4, 8) yielded the highest event-rate ratio and case (4, 16) yielded the lowest event-rate ratio. The best event-rate ratio is 1.49 and the worst was .48. The 16 client

configuration performs better with a large middle cache, a high mean request time and large view pool. The 4 client configuration performs better with a low mean request time, smaller middle cache and the small view pool. The reason for these findings is attributed to the number of hops it takes to find a view at the client level. The fewer clients, the fewer hops, which decreases the dependency among client LPs and ultimately decreases the rollback distance.

It was determined that neither the weight factors nor the mean request time had a significant impact on performance.

5 RELATED WORK

In the area of reverse computation, there has been a great deal of research with seminal work done by Charles Bennett (Bennett 1982). For a detailed summary of reverse computation research for both hardware and software systems, we refer the reader to Carothers, Perumalla and Fujimoto 1999(b).

A number of web-caching architectures and strategies have been proposed, including Abrams et al. 1996; Arlitt and Williamson 1997; Pitkow and Recker 1994; Shi, Watson and Chen 1997; Wessels 1995; and Abrams et. al 1996. The fundamental difference between this work and our caching approach is the complexity of queries. Our view storage system assumes that views are the results of arbitrarily complex SQL queries and not web-pages. Consequently, it is generally sufficient to measure the effectiveness of the caching strategy in terms of hit rate. That measure is insufficient for view storage systems. The reason is because views are typically not the same size, nor the same complexity to produce and because of varying network latencies that will take longer to transmit depending on the location of the source database system. Thus, hit rate is not an accurate indicator of performance.

The parallel synchronization issue has been widely studied. We refer the reader to Bagrodia 1996; Fujimoto 1990; Fujimoto and Nicol 1992; Misra 1986; Nicol and Liu 1997; Richter and Warland 1989.

6 CONCLUSIONS

In this paper we present the design and implementation of a model for a configurable view storage system (CAVES). This model is suitable for execution on a optimistic simulation engine and makes use of reverse computation to support the detection and recovery synchronization mechanism.

Overall, we find that our model performs well with an average speedup of 3.6 on 4 processors over all configurations. Many cases yield super-linear speedup, which is attributed to a slow memory subsystem on the multiprocessor PC. We find that a number of parameters effect key Time Warp performance metrics. In particular, the cache-size effects the rollback distance and the mean request time effects the lookahead of models which in the presence of remote messages will generate more rollbacks. Finally, the number of middle level servers and clients per server appear to increase the dependencies among LPs and thus would result in more rollbacks.

REFERENCES

Abrams, M., C. R. Standridge, G. Abdulla, E. A. Fox and S. Williams. 1996. Removal policies in network caches for world-wide web documents. In *Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 293–305.

Arlitt, M. F., and C. L. Williamson. 1997. Trace-drive simulation of document caching strategies of internet web servers. *Simulation* 68(1): 23–33.

Bagrodia, R. 1996. Perils and pitfalls of parallel discrete-event simulation. In *Proceedings of the 1996 Winter Simulation Conference* eds. J. Charnes and D. J. Morrice. 136–143.

Bennett, C. 1982. Thermodynamics of computation. *International Journal of Physics*, 21: 905–940.

Carothers, C. D., D. Bauer and S. Pearce. 2000. ROSS: A high-performance, low memory, modular time warp system. In *Proceedings of the* 14th *Workshop on Parallel and Distributed Simulation (PADS 2000)*, 53–60.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999(a). Efficient optimistic parallel simulations using reverse computation. In *Proceedings of the* 13th *Workshop on Parallel and Distributed Simulation (PADS'99)*, 126–135.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999(b). Efficient optimistic parallel simulations using reverse computation," (journal version). *ACM Transactions on Computer Modeling and Simulation (TOMACS)*, 9(3): 224–253.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999(c). The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture, In *Proceedings of the 1999 Winter Simulation Conference*. 373–381.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33(10): 30–53.

Fujimoto, R. M. and D. M. Nicol. 1992. State of the art in parallel simulation. In *Proceedings of the 1992 Winter Simulation Conference*, 122–127.

Jefferson, D. R.,. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7 (3): 404–425.

Misra, J. 1986. Distributed-discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

Nicol, D. and X. Liu. 1997. The dark side of risk: what your mother never told you about time warp. In *Proceedings of the 1997 Workshop on Parallel and Distributed Simulation*, 188–195.

Pitkow, J. E. and M. M. Recker. 1994. A simple yet robust caching algorithm based on dynamic access patterns. In *Proceedings of the First International Conference on the World-Wide Web*, <citeseer.nj.nec.com/pitkow94simple.html>.

Richter, R. and J. C. Warland. 1989. Distributed simulation of discrete event systems. *Proceedings of the IEEE* 77(1):99–113.

Rosenblum, M., E. Bugnion, S. Devine and S. A. Herrod. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation* 7(1): 78–103.

Shi, Y., E. Watson, and Y-S. Chen. 1997. Model-driven simulation of world-wide-web cache polices. In *Proceedings of the 1997 Winter Simulation Conference*, 1045–1052.

Wessels, D. 1995. Intelligent caching for world-wide web objects. Master's Thesis, University of Colorado.

ACKNOWLEDGMENTS

This work was supported by NSF Grant #9876932. The authors also appreciate the invaluable effort on the part of Jim Percent in helping to construct an initial version of this parallel simulation model.

AUTHOR BIOGRAPHIES

GARRETT YAUN is a Ph.D. student in the Department of Computer Science at Rensselaer Polytechnic Institute. His interests include parallel and distributed systems, networking and modeling and simulation. His e-mail address <yaung@cs.rpi.edu>.

CHRISTOPHER D. CAROTHERS is an Assistant Professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received his Ph.D., M.S. and B.S. degrees from Georgia Institute of Technology in 1997, 1996 and 1991 respectively. His research interests include parallel and distributed systems, modeling and simulation, networking, and real-time systems. His e-mail address is <chrisc@cs.rpi.edu>.

SIBEL ADALI is an Assistant Professor in the Computer Science Department at Rensselaer Polytechnic Institute. She received her Ph.D. and M.S. degrees from University of Maryland in 1996 and 1994 respectively. Her research interests include information integration, multimedia information systems and query optimization. Her e-mail address is <sibel@cs.rpi.edu>.

DAVID SPOONER is a Professor and Acting Chair of the Computer Science Department at Rensselaer Polytechnic Institute. He received his Ph.D. and B.S degrees from Pennsylvania State University in 1981 and 1975 respectively and M.S. degree from Cornell University in 1978. His research insterest include engineering database systems, object-oriented systems, database security, and Computer Science and Information Technology education. His email is <spoonerd@cs.rpi.edu>.