Computer Science Master's Project

Improving the Performance of Remote Read with Fork and Thread Technology

Author:	Jianliang Yi
Degree:	Master of Computer Science
Submit Date:	March 2002

TABLE OF CONTENTS

ABSTRACT

With the development of the networks, more and more useful data that are needed by local programs are located on remote machines. So remote read is more frequent in current systems. If we can improve the performance of remote read, we can greatly improve the performance of many systems related with operations of remote read.

Many such systems have local caches to temporarily store the remote data. In this project, we use the local cache and checkpointing technologies to improve the performance of remote read by creating two branches at the point of remote read. One branch is called optimistic branch. It'll just read out the necessary data from local cache and continue the local processing. Another branch is called pessimistic branch. It'll first do remote check to see if the data in local cache is valid. If yes, it'll end itself and notify the optimistic branch about the check result to let it continue. If no, it'll stop the optimistic branch and do remote read followed by local processing. We create these two branches by doing checkpointing to local programs.

By the help of checkpointing and local cache technologies, we can greatly improve the performance of remote read in many situations. It's a very useful technology in developing high performance systems related to remote read.

1. Introduction

1.1 Data Replication

Distributed computing systems replicate objects on multiple sites to improve both availability and performance. Replication improves availability by letting users access data even when some sites are non-functional. Replication also improves performance by letting users access data from a nearby site to avoid a remote network access or from an idle site to achieve a better load balance [1, 2, 3].

Data replication introduces a new problem: how to keep all the copies consistent. If read-only pages are replicated, there is no problem. The read-only pages are never changed. So all the copies are always identical. Inconsistency problem is impossible here. But when we replicate read-write pages, if any process attempts to write on a replicated page, a potential consistency problem arises: changing one copy and leaving the others alone is unacceptable.

Maintaining perfect consistency is especially painful when the various copies are on different machines that can only communicate by sending messages over a slow network. In some Distributed Shared Memory systems, the solution is to accept less than perfect consistency as the price for better performance [4, 5]. But in some situation, strict consistency is necessary [6].

In general there are several approaches to keep the data consistency: central control, write update, write invalidate, and so on. Every method has its advantages and its disadvantages, and is suitable in some situations [6].

In the write invalidate strategy, when one copy is modified, all other duplicate copies must be invalidated. When a program tries to access an invalidated data copy, the system will first read the latest version from the original copy. In this strategy, no extra messages are needed when writing to a page continuously. But if another node wants access to this page, a whole page must be sent.

In the write update strategy, when some data on a page is modified, this change must be sent to all of its duplicate copies immediately. In this strategy, all writes need network communication to transfer the modified data (not a whole page). But all reads need no extra cost, just read from the nearby copy.

The write update strategy works better when read accesses occur frequently. Otherwise the write invalidate strategy will be better. We can also select other strategies. To select a suitable strategy is the key point here to improve the performance. But no matter what strategy is selected, remote read is a kind of very important and frequent operation in the system. The performance of remote will directly affect the performance of the whole system. So performance improvement of remote read will be very helpful to improve the performance of the whole system.

1.2 Checkpointing

Checkpointing is the mechanism to save the current system state. The most common use of checkpointing is in fault tolerant computing where the goal is to minimize loss of CPU cycles when a long executing program crashes before completion. By checkpointing a program's state at regular intervals, the amount of lost computation is limited to the interval from the last checkpointing to the time of crash. Another common use of check pointing is in the state saving mechanism in optimistic distributed / parallel simulation [7, 8, 9]. Checkpointing provides a way to rollback so that the simulation can be executed out-of-order [10, 11].

1.3 Using Checkpointing in Remote Read

With the development of the Internet, some new circumstances occur. Let's consider such a situation: There is an original copy of an object located somewhere in the Internet. When a local program wants to access this object, it first downloads it to local cache. Every time when the local program tries to access this object, the fastest way is to read out the local copy in the cache. But this object may be modified since last read. So the local program will first check with the original object to make sure that the local version in the cache is up-to-date. Thus a remote check is always necessary before reading this object. After the remote check, if the local copy is the latest version, the program will directly read out the local version. If not, the program needs to remote read this object from remote machine to local cache again before local execution continues. So before the local execution can go ahead, it must wait for the result of remote check. When the speed of the network is relatively slow, a lot of time is wasted in this kind of waiting.

We need some mechanism to improve the performance in this situation. Write update and write invalidate strategies are not suitable here since it's obvious that the original copy doesn't know how many local copies have been created and where they are. So the only way is that local data should be checked before they are read by local program. Thus the performance of remote check and remote read becomes the critical point in solving this problem. Here we can use the checkpointing technology for this purpose.

With the checkpointing technology, we can make two identical copies of the local program. One of the copies can go ahead directly to read the data from local cache and continue the local execution without waiting for the result of remote check. It just assumes that the data in local cache is valid. If it happens to be true, we actually do the remote check and local processing in parallel, which will greatly reduce the remote check cost.

2. System Design

2.1 Overview of the System

What I'll design in this system is to use checkpointing mechanism to improve the performance of remote read in the circumstance described above.

Here is the general idea of the system. With the help of checkpointing mechanism, we can first make a copy of the local program just before it'll begin to do the remote read. We call one copy the optimistic branch and the other pessimistic branch. These two branches will go ahead concurrently, as described below.

The optimistic branch will assume that the local copy is ok and directly read data out and continue the local execution. In the pessimistic branch, it'll do the regular remote check. This branch will first do the remote check to make sure if the data in local copy are valid. If it is, pessimistic branch will abort itself and leave the optimistic branch alone to go ahead. In this case, it seems like that the program doesn't do any remote check and directly read out the local copy. Of course there are some overheads introduced by checkpointing, but the performance will still be greatly improved in most circumstances. If the data in local cache is found to be invalid, the pessimistic branch will stop the optimistic branch, read the object from remote machine and continue the local execution. In this case, it seems that the local program just do remote check and remote read. Of course we add some overheads to the normal processing.

This is just an overview. There are several problems that we need to solve before it actually works for us. Here are some of them: how to do the checkpointing to local programs efficiently? In what situations can this mechanism help us get better performance? How far can the optimistic branch go before it has to wait for the result of the other branch? Now let's give some analysis to these problems.

2.2 System Architecture

Figure 2.1 is the system model of our test system. In the figure we can see that the whole system comprises many nodes that are interconnected by network. The network can be the Internet or any other networks and each node is an independent entity existing in the network. Each node has two basic functions: provide some useful original data to the whole system and provide connection service to its local clients (programs).

Let's first discuss its first function: provide some useful original data to the whole system. Each node contains some data that are managed by this node. When a node joins the system, it can provide some new source of data to the whole system. Other nodes that have the authorization can visit these data through the interface provided by this node. In other words, all the global data of the system are distributed in all the nodes of the system.

Now let’s discuss the second function of each node: provide connection service to local clients. All the data in this system are provide by independent nodes. How can a local client visit the data it needs that are distributed in the system? Each node provides such functionality. A local program only needs to locate a nearest node in system and connect to it. Then the local programs can access all the global data provided by the system through this node. This node provides the interface to do the access and the local programs don’t need to know the details of data distribution. Local programs can visit all the data just like they are local. There is a mechanism in the node that can help to do the remote check and remote read transparently.

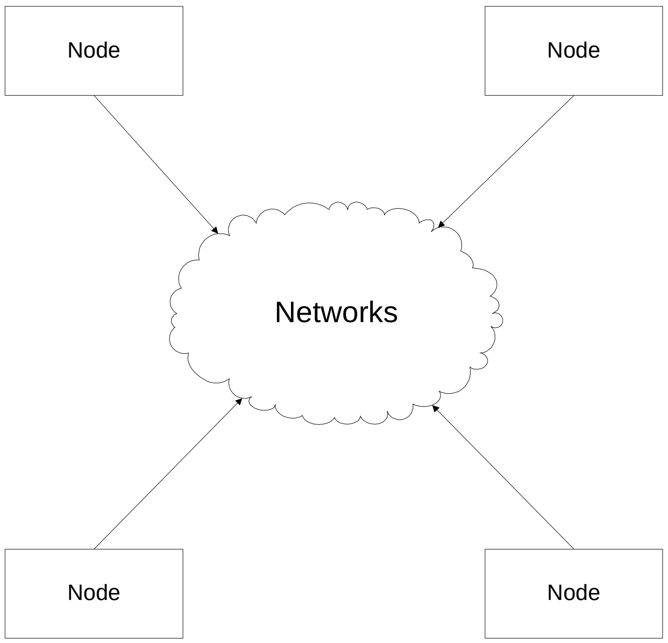


Figure 2.1. System Architecture

Now let’s describe the details in each node. Figure 2.2 shows the architecture of each node. From figure 2.2 we can see that each node has four major components: Data Server, Client Server, Local Cache and Original Data.

Original Data is the data that are maintained by this node. They are part of the global data in the whole system and are managed by the Data Server.

Data Server provides three functions: Check, Read and Write. These functions are provided to both local Client Server and any other Client Servers from other nodes in the system that have the authorization to access the data maintained in this node. All these operations are performed on the Original Data in this node. So we can consider the Data Server as an interface between the Original Data and outside world.

Local Cache is a very important component in this architecture. The first usage of Local Cache is to temporarily store the data read from remote nodes. Since these data are read from remote machine and will be used for local processing in local programs, we need to find a place to store these data for local programs. Local Cache is the place. Yet Local Cache has a more important usage in our system: it provides the necessary data that is needed by optimistic branch to continue local processing without remote check, assuming the data in Local Cache is up-to-date.

Client Server is the most critical component in this system. It connects local programs and Data Servers in all nodes in the system. So it's the interface between local programs and the whole system. When a local program has a data access requirement, the Client Server will first check if the data belong to the local node. If yes, it simply does local memory operation through local Data Server. If no, it'll do remote data operation through networks and Data Servers on other nodes.

If the request is a remote write, Client Server will process it as normal: write the new date to the remote (local) node where the original copy of data is stored. But if the request is a remote read, Client Server will use the checkpointing mechanism to improve the performance of remote read. We'll introduce this procedure of Client Server in 2.3.

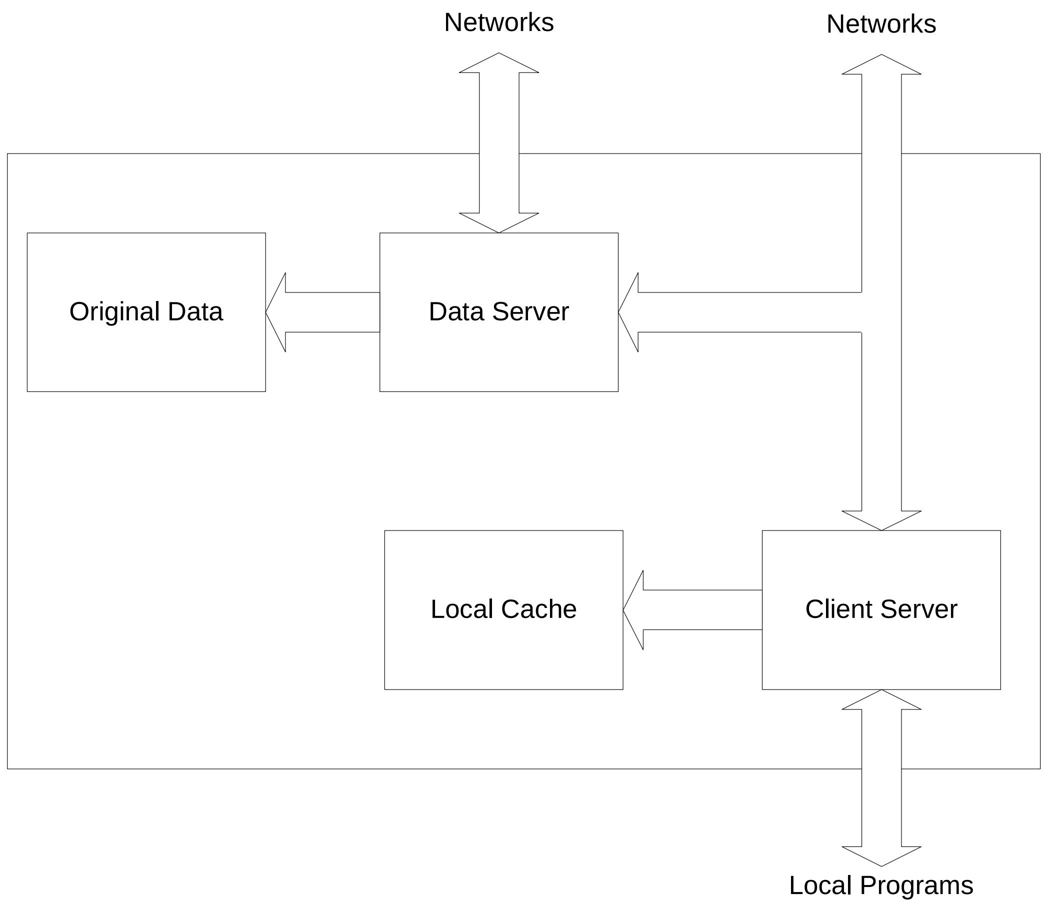


Figure 2.2. Architecture of a Node

2.3 System Flow

Now first let's describe normally how a remote read will be processed. Figure 2.3 below is the procedure of the regular remote read.

When a local program needs to read some data, it'll send a request to Client Server on local node. Client Server will first check if the original copy of the data is on local node. If it happens to be on local node, local Client Server can easily satisfy the request by reading out the data via local Data Server. If the read request is remote, it'll first check if the data in Local Cache is the latest version with the remote node. If it is, then just read out the data from Local Cache. Or else, Client Server will get the latest version of data from remote node by remote read to Local Cache, then resume the local processing.

In this execution flow we can see that if the data is local, the performance is very good. But if the original data is remote, the cost is very expensive, even if the data in Local Cache is valid. In this case, actually the local program can directly read out the data from

Local Cache and continue the local processing. But it has to wait for the result of remote check. If in most cases the data in Local Cache is ok, we can reduce the cost of remote check greatly with our algorithm.

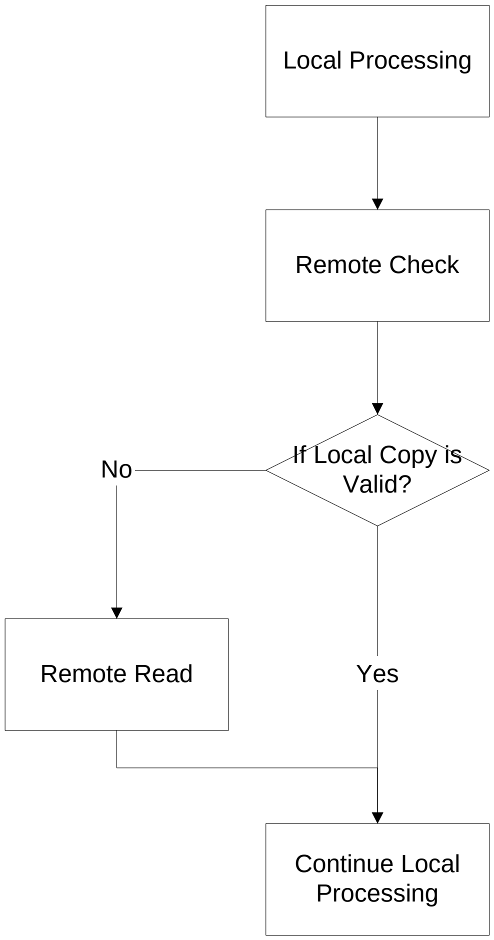


Figure 2.3. Regular Remote Read

Figure 2.4 is the processing flow of remote read with checkpointing mechanism. When a local program sends a remote read request to Client Server, the system will first use the checkpointing mechanism to make a copy of the local program, which means that there are now two identical copies in the system. We call these two branches optimistic branch and pessimistic branch. These two branches will go ahead parallel with some synchronization to make sure the correctness of execution.

In the optimistic branch, the local program will directly read out the data from Local Cache and continue the local processing until it reaches the next checkpoint. We'll discuss this problem in 2.5. This branch will wait at this point if it has already reached this point while the pessimistic branch has not gotten the remote check result yet. After the pessimistic branch gets the remote check result, it'll notify the optimistic branch to go ahead if the data in Local Cache is ok.

In the pessimistic branch, it’ll do the remote check and remote read almost the same as regular remote read, except that it needs to synchronize with the optimistic branch. So the pessimistic branch first does remote check from remote node. Based on the remote check result, two different actions will be taken. If the data in Local Cache is ok, the pessimistic branch will notify the optimistic branch to go ahead, and terminate itself. If the optimistic branch has already waited at the next checkpoint, it’ll resume the execution. If the optimistic branch has not yet reached this point, which means that it’s still doing the local processing before the next checkpoint, this notification will be stored in the system. And when the optimistic branch reaches this point, it’ll go ahead without any wait.

If the data in Local Cache is invalid, the first thing the pessimistic branch will do is to stop the optimistic branch because obviously the optimistic branch is on the wrong direction since the data it reads from Local cache is bad. Then the pessimistic branch begins to do remote read and continue local execution, just as the regular remote read.

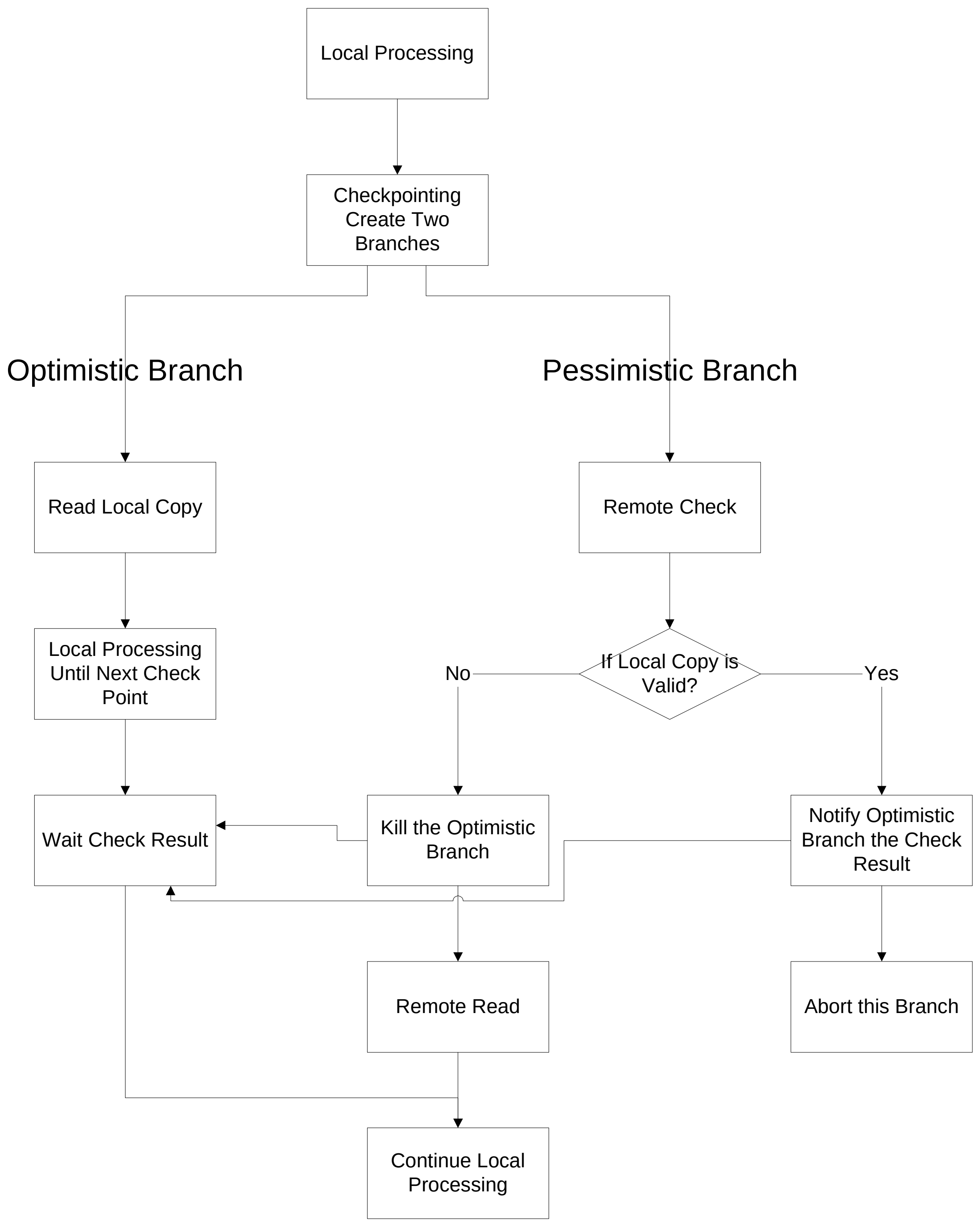

Local Processing
Checkpointing Create Two Branches
Optimistic Branch
Pessimistic Branch
Read Local Copy
Remote Check
Local Processing Until Next Check Point
If Local Copy is Valid?
No
Yes
Wait Check Result
Kill the Optimistic Branch
Notify Optimistic Branch the Check Result
Remote Read
Abort this Branch
Continue Local Processing

Figure 2.4. Remote Read with Checkpointing Mechanism

2.4 Theoretical Performance Analysis

The purpose of our remote read mechanism is to reduce the cost of remote read. It’s very useful but not in all circumstances. Now first let’s do some theoretical analysis to see in what circumstances our mechanism can help us get better performance.

Let's assume that the time it needs to do the remote check is Tc, the remote read time is Tr, the time to do the checkpointing is Tf, the probability that the local copy of the data is invalid is p.

If we ignore all the other cost that may be introduced by our mechanism, we can get that in normal remote read the execution time T1 is:

$$T1 = Tc + p * Tr$$

And when using our mechanism, the execution time T2 is:

$$T2 = Tf + p * (Tc + Tr)$$

When T2 < T1, our mechanism can help local programs to get better performance. So we can get:

$$Tf < (1 – p) * Tc$$

This formula means that when the execution time of checkpointing is shorter than the execution time of remote check times (1 – p), we can get better performance from our system than normal read systems. Tc is a value decided by the network environment. When the network is faster, Tc is smaller; when the network is slower, Tc is larger. But anyway, it's not controlled by our system. Also p is decided by the behavior of programs in the system. In order to get good performance in our system, the only thing we can do is to try to reduce Tf, which is the time spent on checkpointing the local programs. So the main thing we need to do when implement this system is try to provide a good mechanism to do fast checkpointing.

The analysis above is just in the theoretical circumstance. In actual execution, we still have other influence from the system. For example, when we do the checkpointing and begin two branches in parallel, they will share the resource of the local system, which may cause each branch executes a little bit slower than it can occupy all the resources by itself. This means that in order to get better performance, we need to do the checkpointing a little bit faster than the result above to achieve our goal.

Anyway, from the analysis above we can easily see that the critical point in designing and implementing of this system is to do checkpointing faster. We'll introduce the technologies we plan to use to do checkpointing and compare them to each other in 2.6.

2.5 The Next Checkpoint

One difficult point in the design of this system is how to decide the next checkpoint. Let's first define what's the next checkpoint and why we need to consider it. When we execute the optimistic branch, this branch cannot go ahead too far away. That’s because our system is based on such an assumption: if the data in local cache is invalid, we can cancel the optimistic branch, which means that all the operations in optimistic branch can be rolled back.

Some operations in programs cannot easily be rolled back, such as communicating with a remote process, etc. Yes theoretically these operations can be rolled back. For example we can remember all the program states at the checkpoints using state saving algorithm. But practically it’s too expensive and too complicate to do this kind of state saving and roll back. We will lose all our gain in performance obtained by our mechanism. So we must let local programs stop at next checkpoint to wait the result of remote check from the pessimistic branch.

There are two ways to detect the next checkpoint: manually or automatically.

We can automatically detect this kind of checkpoints by the system. The system can monitor all the operations that belong to this kind of checkpoint operations. If one of these kinds of operations occurs, the system will detect that it should be the next checkpoint. The system will let the optimistic branch stop at this point. But this method is very difficult to implement and is very expensive. Since there are so many possible checkpoint operations, we much monitor all of them, which is very inefficient. Also in order to achieve this purpose, we have to merge our system to very lower operating system to get better control and more information about these operations. This inefficiency and inconveniency will lead to the failure of our system.

Another way to do this is by programmers. Our system provides a method or function so that programmers can use this method to apparently give out the next checkpoint in their programs. Thus when the program is running in our system, it can let our system know the next checkpoint easily, almost without any additional cost.

We’ll take the second way in our system. Although we may add a little burden on the programmers who using our system, we can get much better performance which is the original purpose of this system. And also this kind of burden is really small to programmers.

2.6 How to do Checkpointing

We’ve discussed above the importance of doing fast checkpointing. Here we’ll discuss the technologies that can be used to do fast checkpointing. Basically there are two technologies that can easily do this: fork and thread.

In UNIX systems, fork() is a system call to generate a copy of a process. We can use this in our system to do the checkpointing. When we begin to do the remote read in the program, we first call fork() to get two identical copies of the process. One process will go through the optimistic path and the other will go through the pessimistic path.

The advantage of fork() is its clear and simple interface. Since these two copies are two separate processes, they have different data and code space. So they are absolutely independent and the execution of one will not affect the other. So what we need to do is simply doing fork(). That's all.

The disadvantage of fork() is the cost. To generate another process in the system will take relatively longer time while reducing execution time is the original goal of our system. If we cannot do checkpointing quick enough, why we bother to design such a complicated system instead of to do remote read after remote check straight forwardly. So in relatively slow networks, fork() is a very good way to achieve better performance and simple implementation. But in relatively fast networks, its cost is something we want to avoid.

Another way is use the thread mechanism. Thread is also called light-weighted process in some systems. It's within a process. Different threads in a process share the same data and code space, but have different execution controls. We can create optimistic branch and pessimistic branch as two threads. These two threads will go ahead as described above.

The advantage of thread is its lower cost compared with fork(). Since the system doesn't need to create a new process for the new thread, it's much faster to create a new thread. So it's easier to get better performance. Also lower cost means that we can use our mechanism in much faster networks.

But thread also has its disadvantage. Programmers who implement the system need more complicated design and implementation. Also users of the system need to do something in order to help the system to achieve its goal. That's because two thread share the data, which means that the execution in one thread will influence the other. When the optimistic branch goes ahead and makes some modification to the data, and it turns out that the data in Local Cache is invalid, we need to roll back to the checkpoint and redo the execution based on the new data from remote machine. This means that the system must store the system state at checkpoint and restore the state if needed.

If we want the system to be transparent enough, we can just save everything of the program so that we will not bother users to provide any information. But if we do so, the cost is just as the fork() above, which means thread loses its advantage. The only way to get back its advantage is that users should help the system to figure out what's really needed to be stored at checkpoint. Thus the system can only store the necessary information to get high performance and lower cost. But users have to have some knowledge of system and know that what may be modified by the optimistic branch so that the system need to store these data before the optimistic branch begins.

From the analysis above we can get such a conclusion: If we want to use our mechanism in fast networks, thread is better. If in slow networks, fork() is better since we can get better performance together with the simplicity. For system programmers, thread is a better choice since its better performance. For normal users, fork() is a better choice for its clear interface.

3. System Implementation

3.1 Implementation with Fork() in Java

As we all know, in UNIX systems, there is a system call named fork() [12]. A process can call fork() to generate a new process. The new process, which is called child process, is a duplicate copy of the original process. The original process is also called parent process. The child process has its own data space and stack.

Different UNIX systems have different implementation of the fork(). Since it's quite common that there is an exec() following the fork(), some UNIX systems don't do the copy of data space and stack when the fork() is call. Instead, they use the Copy-On-Write (COW) technology. Only when the process try to modify the data in these areas will the real copy be made.

Since the different implementation of the fork(), the performance of fork() in different systems is also different. We test the speed of fork() in Sun and Linux systems, and find that the fork() in Linux system is almost ten times faster than in Sun system. So we can implement our system with fork technology in Linux system while can only use the thread technology in Sun in order to get good performance.

Another import point we need to notice is how can we call fork in Java [13]. We can use the Java Native Interface (JNI) to access non-Java code from Java programs. First we need to define the non-Java code interface in Java programs with the key word "native". Then we can use the command "javah –jni" to generate C language head file. We need to include the head file generated by javah into the C code that will be called by Java programs. Please see the attached source code for details.

The implementation with fork() technology is relatively simple compared with the implementation with thread technology. When the program needs to do checkpointing, it can simply call the fork(). The fork() will generate two identical processes, one parent and one child. The parent process is the optimistic branch and the child process is the pessimistic branch in our model. The optimistic branch will go ahead to read from Local Cache and continue the local processing until the next checkpoint. The pessimistic branch will do the remote check from remote machine to see if the data in Local Cache is valid. If yes, the child process (pessimistic branch) will stop itself and notify the parent process (optimistic branch) the result. If no, the child process will kill the parent process and continue to do remote read and local processing.

One difficult problem here is how to detect where the next checkpoint is. We can either get this information from programmers or from running systems.

It’s easy if programmers can provide the information about the next checkpoint. The implementation of the system will be simple and the performance will be better. The disadvantage is that the programmers have to have the idea where the next checkpoint will be. It needs programmers to involve more into the system. But generally speaking, it’s still reasonable since this kind of information is not very difficult to determine by programmers. We implement our system with this method. We provide an interface (a method) to programmers so that they can call it before the next checkpoint to notify the system to stop before it if necessary.

Also if we want the system to be completely transparent to users of the system, we can implement a daemon in the system to monitor all the sensitive operations, such as remote write. Since it’s too complicated to recover these kinds of operations, when the programs meet these operations, they’ll stop at this point in the optimistic branch and wait for the result of remote check. It’ll be more convenient to programmers who use this system, but the cost of monitoring will lower the performance.

3.2 Implementation with Thread in Java

Another way to implement the system is using the thread in Java. As we’ve mentioned above, in most systems thread will provide better performance than fork(). But the implementation will be more complicated also. That’s because different thread in a process share the same data space and other resources. So if we implement the optimistic branch and the pessimistic branch with thread, they are actually sharing something and are not completely independent.

The implementation of the system with thread looks like this: at the remote read point the program will create two threads, one as optimistic branch and the other as pessimistic branch. But before the two branches begin, the program needs to store the useful status of the program so that the program can roll back to the checkpoint status when the remote check shows that the data in Local Cache is invalid. After backup the necessary status of the program, two branches will go ahead as described above. The optimistic branch will go ahead to do the local processing until the next checkpoint while the pessimistic branch will do the remote check. If the data is valid, the pessimistic branch will stop itself and notify the optimistic branch the result. If the optimistic branch is waiting at the next checkpoint at that time, this notification will make it continue. If the optimistic branch is still running, the system will record this notification so that when the optimistic branch runs to this point, it’ll continue without any wait. If the data in Local Cache is invalid, the pessimistic branch will stop the optimistic branch, load in the stored status to roll back to the checkpoint and do remote read.

From the description above we can see that the key point in this implementation is how to do the backup and rollback quickly. One way is to store all the information about the program. The problem is that it’s almost the same as fork() so that we cannot get better performance from thread while making the system more complicated. Thus fork() may be a good choice.

But in this implementation, we can only record part of the program statuses that are really necessary. But what is needed and what is unnecessary? It’s difficult for the system to determine while programmers are a good source of this kind of information. So the system needs programs to provide two methods (functions), one is used to save the necessary statuses and the other is used to restore them.

In order to get high performance with thread, programmers who use the system need to know more about the system and do more in programming, comparing with the implementation with fork() technology.

These two implementations have their advantages and disadvantages. Which is better really depends on users and the usage of the system.

4. Performance Results and Analysis

We will test our system in two typical cases. In the first case, we do remote reads from a non-changed remote object that has already in the local cache. In this case the optimistic branch will always succeed and the pessimistic branch will always kills itself after it does the remote check. Also there are enough local processing needed between two continuous remote reads so that the optimistic branch can go ahead far enough during the period when pessimistic branch is doing the remote check. This is actually the best performance our system can get for the remote read.

The second case will imitate the real behavior of local programs. The local processing time between two remote reads may be shorter than needed to get good performance. There are remote writes after several rounds of remote reads and local processing. In this case the execution time of local processing and the frequency of remote write may affect the performance of our system greatly.

4.1 Test Case 1

As described above, in this case, the test program should meet two requirements: 1. No write to the remote object so that the data in Local Cache is always valid; 2. There are enough local processing time between two remote reads. We use the small test program described as below:

```
Remote Read();
Loop (n times)
    Remote Read();
    Local Process();
End Loop
```

The first remote read before the loop is used to make sure the data in Local Cache is valid when the loop begins.

Here we test the running time of remote reads while actually always having the latest version of data in Local Cache. We use three methods to do this kind of remote reads:

1. NR (Normal Read): a remote read always contains a remote check followed by a local read (since the local copy is fine)
2. TR (Read with Thread Technology): checkpointing with thread technology to create optimistic branch and pessimistic branch. The pessimistic branch will always kill itself after the remote check while the optimistic branch can go ahead directly without the result

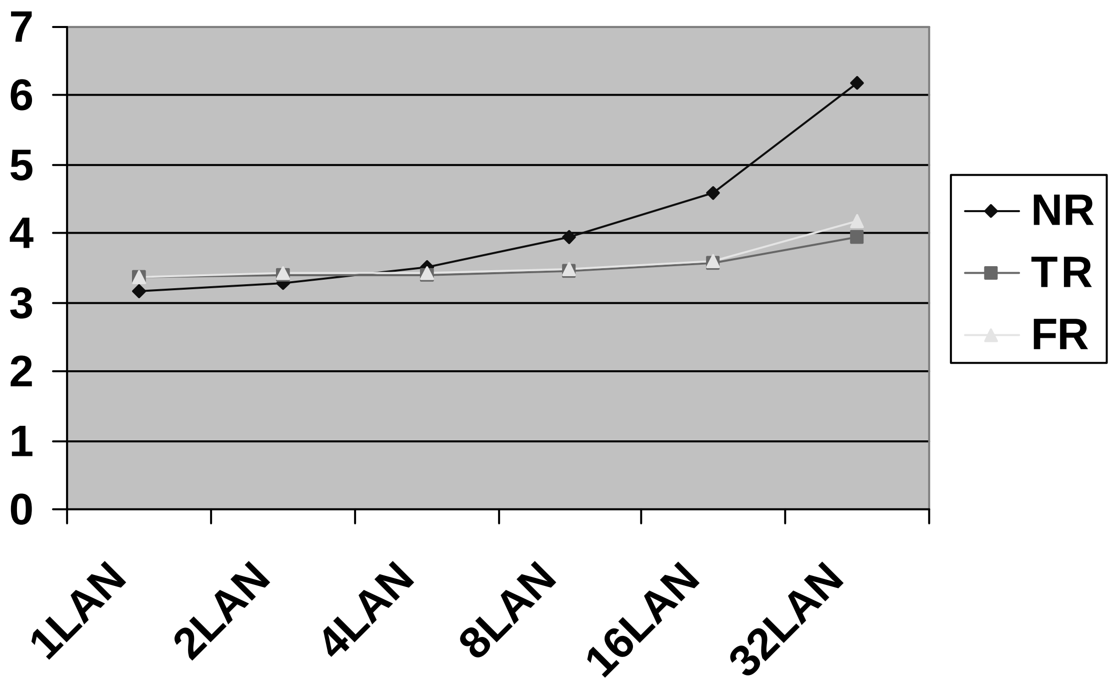
of remote check. So the remote check and local read can be processed concurrently.
3. FR (Read with Fork Technology): checkpointing with fork() technology to create optimistic branch and pessimistic branch. The pessimistic branch will always kill itself after the remote check while the optimistic branch can go ahead directly without the result of remote check. So the remote check and local read can be processed in parallel.

One thing we need to consider is the execution time of local processing between two remote reads. We should make it long enough so that we can guarantee that the optimistic branch will not stop to wait for the result of remote check at the next checkpoint in pessimistic branch. We will select the local processing time to be 30ms and 60ms.

All the experiments are run on cluster machines (ba?.cs.rpi.edu) with Linux system. The size of the remote object is 10K.

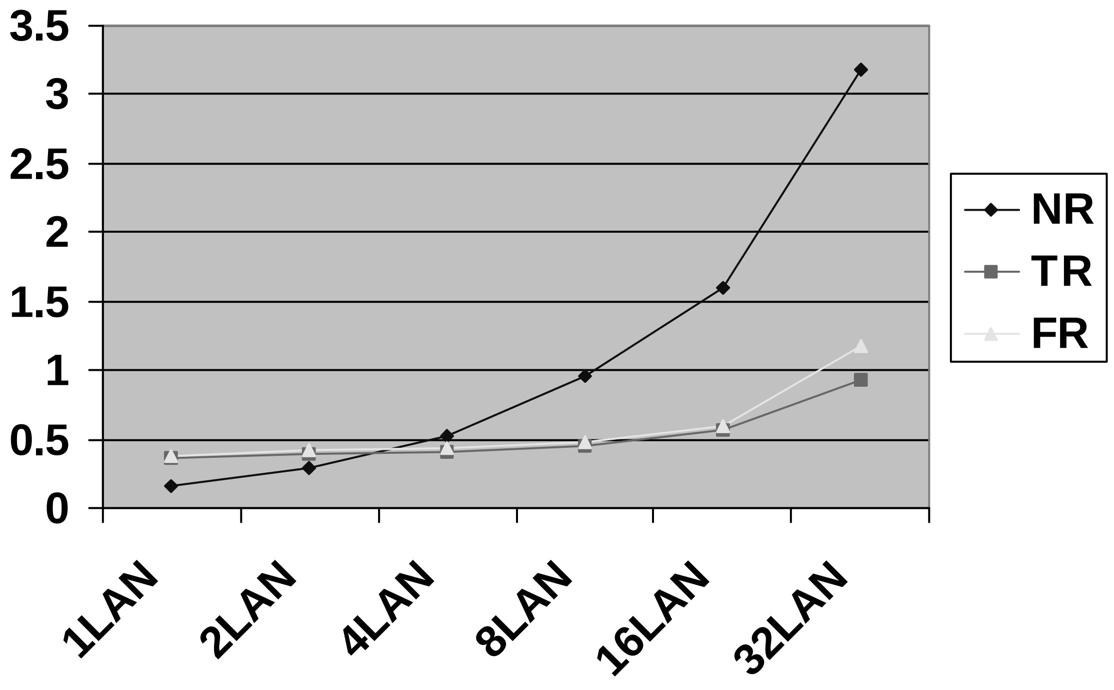
1. Local Process = 30ms, Run 100 times

Here “Local Process = 30ms” means the execution time of local processing between two remote reads is 30 milliseconds. In the graph below x-axis represents network speed. 1LAN means the speed of the LAN in the cluster. We use software to slow down the speed of the network so that we can also test the performance of remote read in relatively slow networks. So here 2LAN means the speed is two times slower than the speed of LAN, 4LAN means the speed of network is four times slower than the spend of LAN…



Graph 4.1

In graph 4.1, the y-axis represents the total execution time of 100 loops in local program, including local process time, which is totally 3 seconds in this experiment.



Graph 4.2

In graph 4.2, x-axis is the same as graph 4.1. But the y-axis represents the total execution time of 100 remote reads, excluding local process time. So we can see clearer in this graph than 4.1 about the performance of remote read in different technologies.

From these two graphs we can see that when the speed of network is about 4 times slower than the speed of LAN, the execution time of remote reads of three different methods are almost the same. When the network speed is faster than that, normal read is better. When the network speed is slower than that, read with fork or thread technologies is better.

This is easy to understand. Still remember the performance analysis in Section 2. We have the equation $Tf < (1 – p) * Tc$. Here p is equal to 0.00 since the probability that the local cache is invalid is not possible in this case. Thus in order to get better performance with fork or thread technology, the system needs to satisfy the condition of $Tf < Tc$ (Tf is the execution time to do the checkpointing while Tc is the execution time of remote check). When the networks speed is very fast, $Tf > Tc$. So the performance of remote read with thread or fork is not as good as normal remote read. When the network speed is slow, we can get Tf < Tc. In this circumstance, the performance of remote read with thread or fork technology is faster than normal remote read.

We can consider Tf = Tc at the point 4LAN. When the network speed is 4 times slower than LAN, the checkpointing time is almost the same as the remote check time. If we can reduce the Tf, we can move this point to the left, which means that our system can be used in faster networks.

Also from the graphs we can see that when the network speed becomes slower, the normal read execution time increases very faster due to the longer execution of remote check. But with fork or thread technology, the read execution time grows very slow. That's because with fork or thread, we can utilize the parallelism between remote check and local processing to greatly reduce the waiting time of remote check. We can see from the graph 4.2 that when the network speed is 32 times slower than LAN, normal read will use more than 3 times execution time than the read with fork or thread technology.

We can see another trend in graph 4.2 that the execution time of read with fork or thread grows very slow at the beginning. But after certain point, it begins to grow faster. In graph 4.2, we can see this trend when the network speed is 16 times slower than LAN. This is because Tc is longer in slow networks, which means the optimistic branch need more local processing between two remote reads in order to prevent the stop for waiting for the remote check result. In this case, the local processing time is 30 ms. When the network is too slow, after the optimistic branch finishes this period of local processing, it has to wait for the remote check result. So the execution time of remote read begins to grow faster.

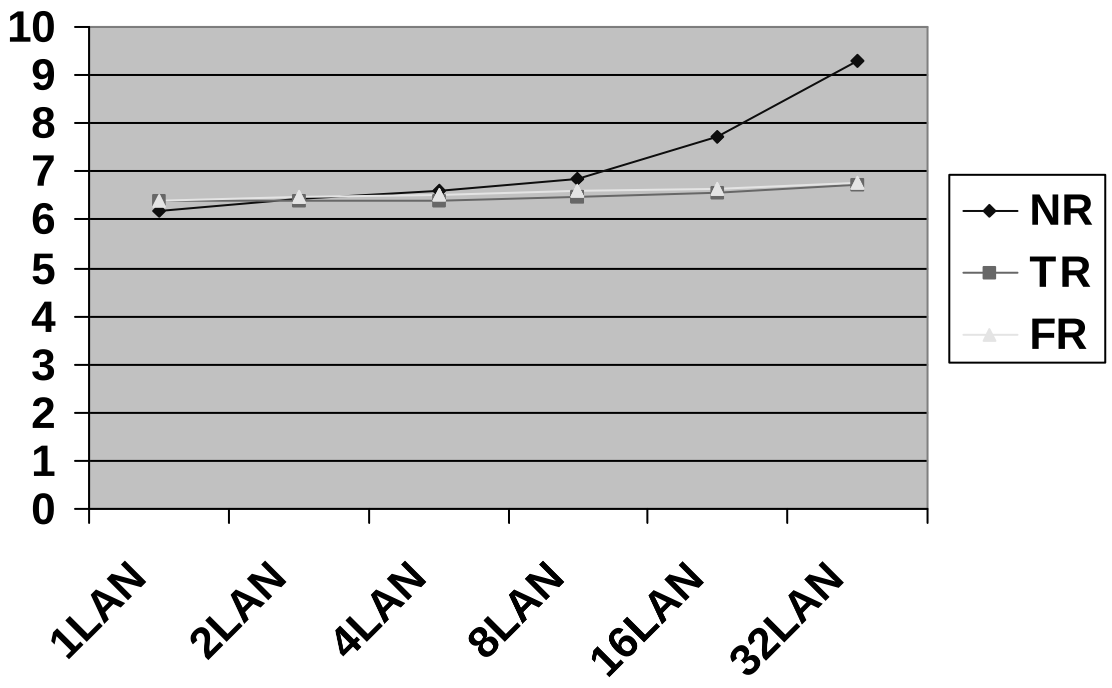
2. Local Process = 60ms, Run 100 times

In this case the only difference is that we change the local processing time to 60ms. All other parameters are the same as above.

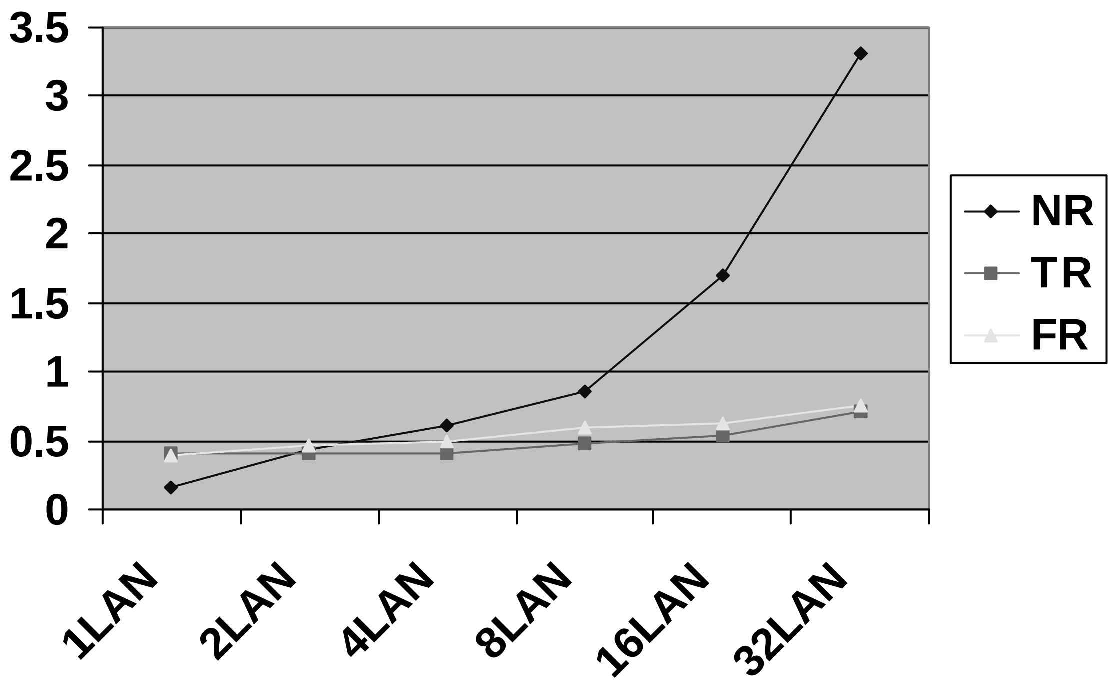
We can see from the graph 4.3 and 4.4 that the trends are almost the same as above. But we can also see an obvious difference here: in graph 4.2, when the network speed is 16 times slower than LAN, the execution time of read with fork or thread begins to grow faster than before. We have explained above that's because the local processing time between two reads is not long enough. But in graph 4.4, there are no such trends. The execution time of read with fork or thread technologies always grow very slow.

That's because the local processing time in this experiment is longer than the case above (changed from 30 ms to 60 ms). This local processing is long enough to prevent the optimistic branch to stop for the result of remote check in our test range of network speeds.

But this point still exists. It’ll occur when the network speed is even slower.



Graph 4.3



Graph 4.4

4.2 Test Case 2

As we have mentioned in the theoretical analysis, if we want to get better performance in our system, we should have Tf < (1 – p) * Tc. Let's assume that Tf is a constant in a certain system. In a fast network, the value of Tc is small. In order to satisfy this condition, we should have small p value, which means that the probability that the data in Local Cache is invalid should be very low. In a slow network, the value of Tc is relatively large. So in order to satisfy this condition, the value of p can be relatively large, which means that our system can fit in more circumstances in slow networks.

In this case, we'll do experiments to verity the analysis above. We use the following local program as the test program:

```
Loop (n times)
    Randomly Do a Remote Write();
    Remote Read();
    Local Process();
End Loop
```

Here we randomly do a remote write before the remote read to make sure that the remote read following the remote write needs to load the necessary data from remote machine instead of Local Cache. In this test case we'll adjust the probability to do the remote write and analyze the different results in different circumstances.

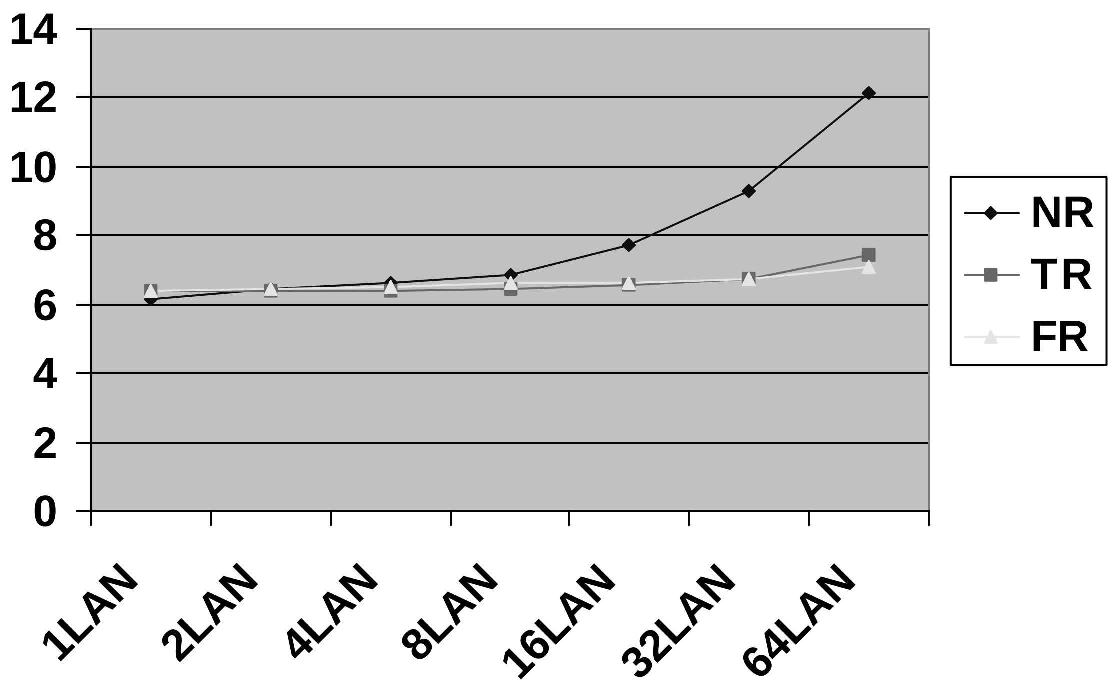
In all test cases in this section, the local processing time is 60 ms.

1. p = 0.00

p = 0.00 means that all the remote read can be finished by reading from Local Cache. If we use the formula above, we get Tf < Tc. This means that when the checkpointing time is smaller than remote check time, system can get better performance than normal remote check plus local read.

From graph 4.5 we can see that when the network speed is about 2 times slower than LAN, the execution time of normal read, read with fork technology and read with thread technology are almost the same. When the network speed is slower than that, we can get better performance in our system.

Also we can see from the graph that when the network speed is very slow (say 32 times slower than LAN), the performance of read with thread or fork technology is much better.



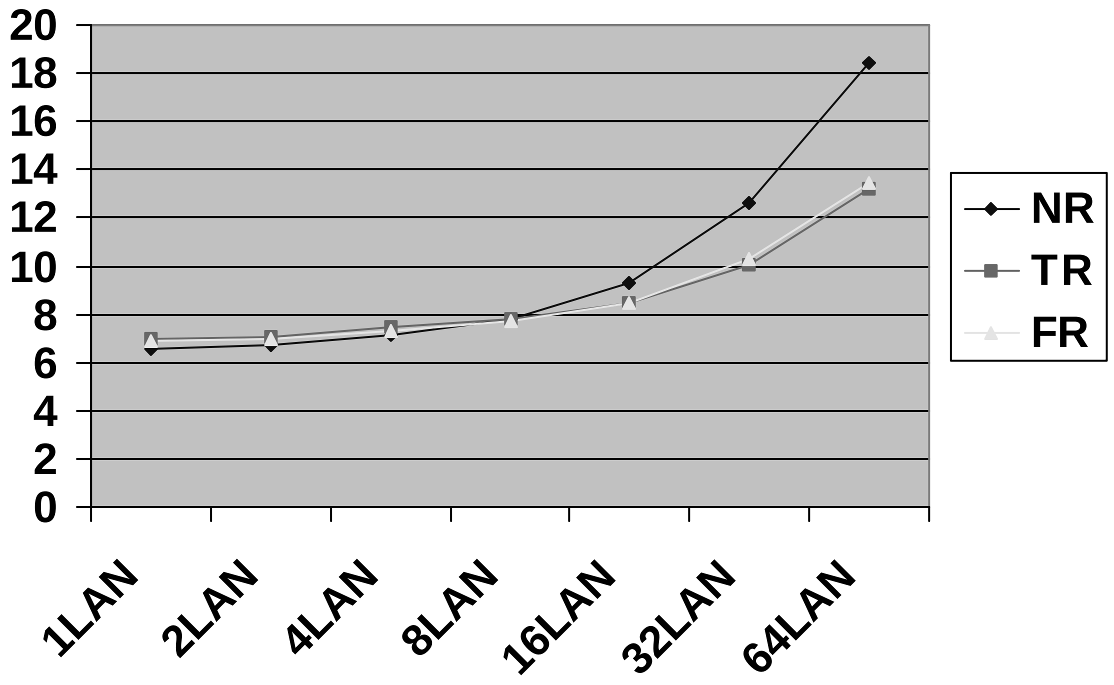
Graph 4.5

2. p = 0.10

p = 0.10 means that when we do n remote reads, in 10% of the total number of reads, the data in Local Cache in invalid and we need to read the object from remote machine. If we still use the formula above, we get Tf < 0.9 * Tc. In a certain system we can consider Tf is fixed. So in order to satisfy the formula, we have to use slower networks than that in the case p = 0.00.

From graph 4.5 we can see that when the network speed is about 8 times slower than LAN, the performance of normal read and read with thread or fork technology almost have the same performance. When the network speed is slower than that point, our system can get better performance than normal read.

When the network speed is quite slow, we still can get very good performance, although from the graph we can see that what we can get from our system is not as much as when p = 0.00.



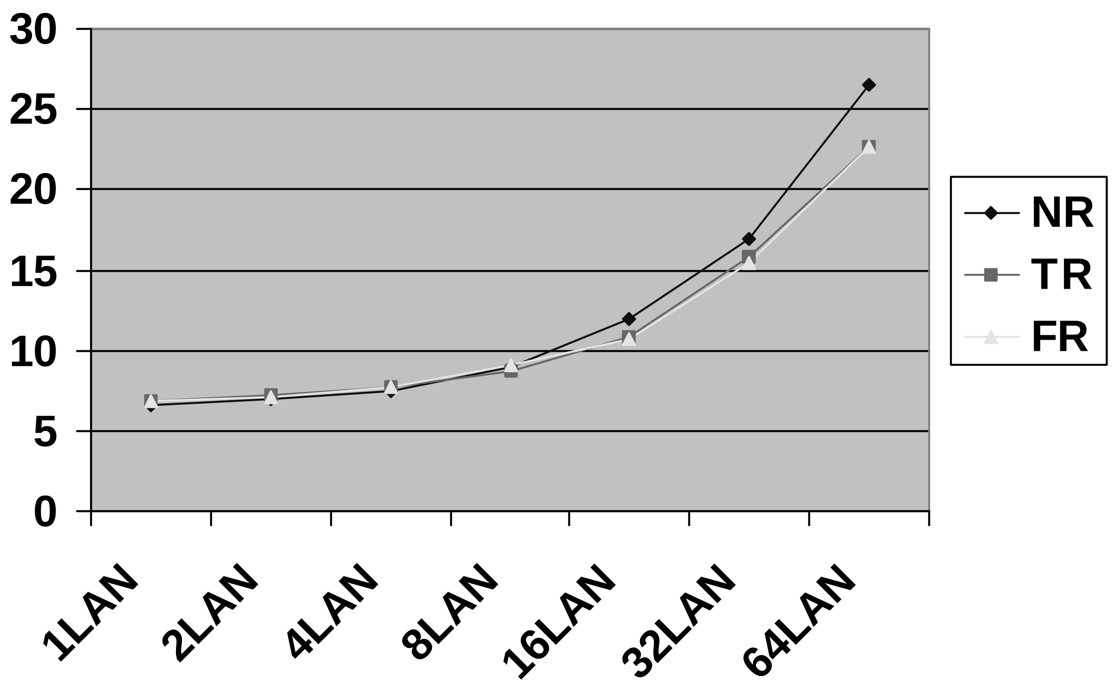
Graph 4.6

3. p = 0.25

p = 0.25 means that when we do n remote reads, in 25% of the total number of reads, the data in Local Cache is invalid and we need to read the object from remote machine. If we still use the formula above, we get Tf < 0.75 * Tc.

From graph 4.7 we can see that when the network speed is quite slow, the performance of read with thread or fork technology has better performance than normal read. But it's obvious that the difference is quite small, which means that actually we cannot get too much from our system.

This is very easy to understand. In our system, when the data in Local Cache is invalid, the pessimistic branch will kill the optimistic branch that reads the data from local cache and continue the local processing. Then the pessimistic branch still needs to do remote read and local process again. So the total cost is large than normal read since we still have the checkpointing expense. So when the percentage that local data is invalid is high, we cannot get much from our system.



Graph 4.7

5. Conclusion

From the results of experiments above we can see that in many circumstances remote read with thread or fork technology can get pretty good performance than normal remote read. Here are some of the factors that will affect the performance of our mechanism:

The first one is the network speed. Since the cost of checkpointing is not very cheap, in order to get better performance, the network speed should not be too fast. For example, LAN is too fast for this technology, but the Internet is a good environment to use our mechanism to get better performance.

The second is the execution time of local processing between two remote reads. If the behavior of programs is one remote read after another without many local processing, we cannot get much from this mechanism. Actually it'll slow the execution down. But generally there will be some local processing between two remote read, so in most cases we can use this mechanism to get better performance.

The third is the percentage of read that the data in local cache is invalid. If the data in local cache are always invalid, we always need to do remote read. Why do we bother to use our mechanism? But in some applications, the frequency of update to the remote objects will not be that frequent. So the percentage of actually remote read is not very high. In this case, we can get much better performance using our mechanism.

Also thread technology and fork technology have their own advantages and disadvantages. In some systems fork can has almost the same performance as thread, for example the Linux system. In these systems, fork technology is a better choice. Since it'll be simple and clear to implement the system with fork technology.

But in some system, for example the SUN system, the fork operation is very expensive comparing with thread. In these systems, we'd better choose thread to get better performance, although the system will be more complicated.

ACKNOWLEDGEMENT

I would like to thank Prof. Szymanski for giving me the original idea of the system and supervision during the whole period of my master project. I would also like to thank Houda Lamehamedi and Qiuju Gu for their advice.

BIBLIOGRAPHY

[1] Yasushi Saito, *Optimistic Replication Algorithms*. International Symposium on Distributed Computing, 2000.

[2] M. Nibhanapudi, B. Szymanski, *Adaptive Parallelism On A Network of Workstations*, High Performance Computing Systems and Applications}, J. Schaeffer (ed.), Papers presented at HPCS98, Edmonton, Canada, May 20, 1998, Kluwer Academic Publishers, Reading, MA, 1998, pp. 439-452

[3] M. Nibhanapudi, B. Szymanski, *BSP-based Adaptive Parallel Processing*, In *High Performance Cluster Computing*, vol. I, Architectures and Systems, Rajkumar Buyya (editor), Prentice Hall, New York, 1999, pp. 702-721.

[4] Boleslaw Szymanski, Gang Chen, Houda Lamehamedi and Arjun Vargun, *Web-Enabled and Speculative High Performance Computing*, Proc. Int. SGI User's Conference, SGI2000, Cracow, Poland, October, 2000, AGH Press, Cracow, 2000, pp. 75-90.

[5] Andrew S. Tanenbaum, *Distributed Operating Systems*. Prentice-Hall Inc., 1995.

[6] Xie Li and Yi Jianliang, *A model for intelligent resource management in a large distributed system*. SCIENCE IN CHINA (Series E), Vol.41 (1), February 1998.

[7] Ewa Deelman and Boleslaw K. Szymanski, *Breadth-First Rollback in Spatially Explicit Simulations*, Proc. PADS97, 11th Workshop on Parallel and Distributed Simulation, Burg Lockenhaus, Austria, June 10-13, 1997, pp. 124-131, IEEE Computer Society, Los Alamitos, CA.

[8] Ewa Deelman and Boleslaw K. Szymanski, *Continuously Monitored Global Virtual Time*, Proc. Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97, Las Vegas, NV, June 30-July 3, 1997, Vol. I, pp. 1-10, CSREA, 1997.

[9] Ewa Deelman and Boleslaw K. Szymanski, *Dynamic Load Balancing in Parallel Discrete Event Simulation for Spatially Explicit Problems*, Proc. 12th Workshop on Parallel and Distributed Simulation---PADS98, Calgary, Canada, June 1998, IEEE Computer Society Press, Los Alamitos, CA, pp. 46-53.

[10] Christopher D. Carothers and Boleslaw K. Szymanski, *Checkpointing Multithreded Programs*. Dr. Dobb's Journal, to appear.

[11] Boleslaw K. Szymanski, Yu Liu, Anand Sastry, and Kiran Madnani, *Real-Time On-Line Network SimulationGenesis,* Int. Workshop on Distributed Simulation and Real-Time Applications DS-RT2001, IEEE Computer Society Press, Los Alamitos, CA, 2001, Cincinnati, OH, August 13-15, 2001, pp. 22-29

[12] Richard Stevens, *Advanced Programming in the UNIX Environment*. Addison Wesley Publishing Company, 1992.

[13] Bruce Eckel, *Thinking in Java*. Prentice Hall Inc., 1998.