Spider web inspired Dynamic Memory Allocation

Siddharth Gurdasani, Wenzhu Liu

Introduction

Spiders in particular orb web spiders, utilize the threads as an external sensory organs to learn about the spatial structure of the environment i.e., where the prey will be falling[2]. And based on this it performs the smart move of thread pulling using its legs (as it is sitting in the center of the web) which in turn creates tension in the thread and aggravates the vibration transmission capacity of the thread. In this way, spiders which are classified into sit and wait predators optimize their prey detection efficiency and adapt according to the environment[3].

Spatial learning performed by the spider can be applied to optimize the dynamic memory allocation algorithms like BBSLR, in order to utilize the resources efficiently and be able to serve maximum number of simultaneous clients.

Static Memory Allocation vs. Dynamic Memory Allocation

Cache memory is a smaller and faster memory which stores the data repeatedly requested by the clients. Searching data in cache memory is time saving because of small size of the memory.

Static memory allocation works on the principle of allocating fixed amount of cache memory (also buffer memory) to the client based on the assumption that the system is in a fully-loaded state. But when the system is not in a fully-loaded state, it might allocate a larger chunk of memory than necessary. And so it cannot use memory efficiently.

While dynamic memory allocation, considers the memory requirement of the clients and reclaims the rest of the memory (when not in use). So dynamic memory allocation calculates the cache memory based on the need of the data at run time and also manages memory more efficiently.

Buffer sharing serves multiple clients by caching data in the memory. It works as follows: if two clients are requesting for the same file around the same time, then the data delivered to the first client can be retained in the memory until it is delivered to the second client. Buffer sharing can reduce memory requirement, enhance system performance and also reduce start-up delay (efficient to search in cache memory). Some algorithms that implement buffer sharing are:

1. **CBS (Controlled buffer sharing)**
   CBS presents a concept of distance threshold d_t, which is the number of buffer blocks which can be used to cache the sharing data. If the needed buffer blocks for the same file when two clients are simultaneously requesting exceeds d_t, then they are not allowed to share the memory. This algorithm computes d_t at system design time i.e., the data will not be cached in the buffer until the new request arrives whose distance does not exceed d_t with the former request. So there will be a transition state when the new request arrives for the same file, and the former request is being cached. And this will result in more

requirement of resources as the new request must be served by some disk memory, until the data is cached. In this way, this algorithm not only expends more resources but also cannot reduce start-up delay (time to respond) due to the transition state.

2. **MUPIC (Multi Policy Integrated Cache)**
   MUPIC mainly depends on comparison of the ratio of cache usage (M_i/M) and main memory usage (B_i/B) to determine whether or not to cache the data. Here M denotes the size of the total memory and M_i denotes the size of the needed memory when a new request is served from the cache memory, while B_i and B are similar to M_i and M but here the new request is served from the main memory. When the ratio of cache usage is lower than the ratio of main memory usage, the request is served from the cache memory. When the two ratios are equal, it does not matter which resource is used for serving the request. Similar to CBS, MUPIC does not cache the data in the buffer until new request arrives which needs to be served with the buffer. So although, MUPIC decreases the consumption rate of a single resource, but it is possible that one resource has reached its limit while another has many available. And the reason for this is, it does not consider the rate of whole resources usage and so if the latter request is served by the limited resource then MUPIC needs to reallocate resources before the request can be served. This results in more start-up delay.

3. **FCFS (First Come First Serve)**
   FCFS maintain queues of requests and selects the queue with the oldest request that is the first one. It is basically a simpler algorithm which does not use buffer sharing and just serves the client according to their time of request.

4. **BBSLR (Balanced Buffer Sharing of Limited Resources)**
   A novel algorithm proposed by Li et al (2007), Balanced Buffer Sharing of Limited Resources (BBSLR)[1] allocates memory dynamically to the clients based on available cache and main memory. It also adjusts the value of distance threshold at run time according to the request distribution of the clients and available memory. In this way, it manages memory more efficiently and also maximizes the number of simultaneous clients. It also reduces average start up delay (response time of the server) by caching the data initially, so that more clients can be served. It further reclaims the cache memory when no longer in use or the service is terminated and thus maintains equilibrium between the main memory resources and cache memory resources. So this algorithm would induce even consumption of memory from both the resources. (Fig 1 and 2)

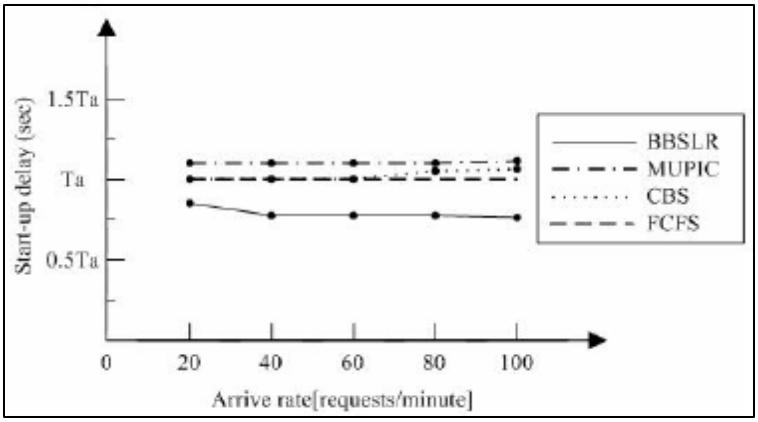

Fig1: Comparison of the response time (start-up delay) of above mentioned algorithms vs. different request arrival rate (requests/minute) by the simultaneous clients

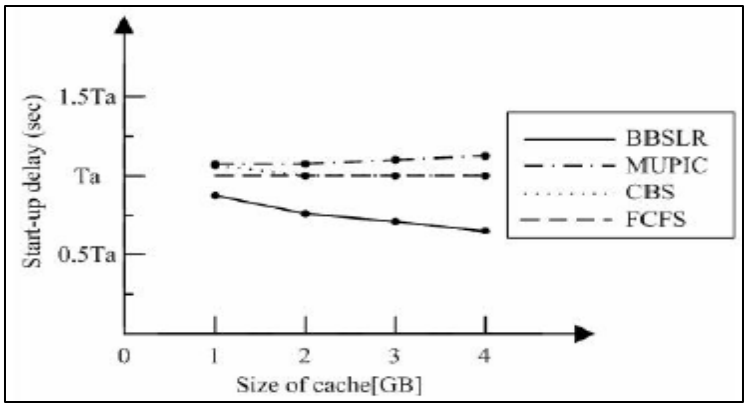

Fig2: Comparison of the response time (start-up delay) of above mentioned algorithms vs. different size of the cache memory by the simultaneous clients

Reclamation algorithm

Here we calculate the time threshold ***t***, which is the limit of interval time between two simultaneous clients requesting the same data. So if the later clients request arrival time – former clients request arrival time is greater than t, then the data is reclaimed from the cache memory and the new request is served by main memory; otherwise it remains in the cache memory. In addition to this, if the cache memory contains some requests that are not asked again by the clients (unique ones), then also they are reclaimed from the memory. Thus the cache memory is dynamically changing according to the size and the time interval of requests.

$$t = \frac{n^4}{M^{\frac{3}{2}} \times 10^8}$$

M – Size of the main memory

n – Number of requests

t –time threshold

D-maximum time interval between two requests for the same data (file)

C-size of the cache memory

We changed the size of the main memory, cache memory and number of requests to determine distance threshold.

Table 1: Trials to calculate the time threshold

Parameters	Trial 1	Trial 2	Trial 3	Trial 4
M(MB)	10000	10000	1000	100
C(MB)	1000	1000	100	10
n	10000	1000	1000	100
D(Seconds)	325.0604	0.0413	0.4982	0.0024
t(Seconds)	100	0.0100	0.3165	0.0010

Dynamic memory Allocation vs. Spider

As the BBSLR allocates memory dynamically depending on the request distribution and available memory, spider also varies tension levels on the radial threads according to the frequency of preys falling on either side of the web. In the Table 2, we have demonstrated the correlation between the static model (Static memory allocation) and dynamic model (dynamic memory allocation) inspired by the spider, where the requests by the clients can be compared to the falling preys and accordingly, the memory allocated can be compared to tension levels of the threads.

Table 2: Comparison of memory allocation in static model vs. dynamic model

Static Model			Dynamic model inspired by spider		
Clients	**Requests**	**Memory allocated**	**Clients**	**Requests**	**Memory allocated**
Client 1	**1**	**5**	**Client 1**	**1**	**1**
Client 2	**1**	**5**	**Client 2**	**1**	**1**
Client 3	**4**	**5**	**Client 3**	**4**	**5**
Client 4	**8**	**5**	**Client 4**	**8**	**9**
Client 5	**4**	**5**	**Client 5**	**4**	**5**
Client 6	**6**	**5**	**Client 6**	**6**	**7**
Client 7	**7**	**5**	**Client 7**	**7**	**8**
Client 8	**7**	**5**	**Client 8**	**7**	**8**
Client 9	**2**	**5**	**Client 9**	**2**	**2**
Client 10	**4**	**5**	**Client 10**	**4**	**5**

In the Table 3, we are explaining the time difference in searching from the main memory and the cache memory. Also, the cache memory is changing dynamically (based on the available

memory and time interval t) and so some of the data not requested by the clients is reclaimed. Due to this, there is an equal consumption of both main memory and cache memory. So due to reclamation of memory from cache, it can simultaneously serve more clients as well as can store more data and eventually lead to faster searches. This makes the server client network robust.

Table 3: Depicting the time difference in searching from main memory vs. cache memory (changing dynamically)

No	Time to load from main Memory(milliseconds)	Time to load from Cache Memory(milliseconds)
1	0.0028610	0.0019073
2	0.0030994	0.0028610
3	0.0040531	0.0009537
4	0.0030994	0.0019073
5	0.0030994	0.0011921

This time also includes the print time on the console but without it also the time difference exists.

Conclusion

As the spider performs spatial learning of the environment, server client request management system can be optimized based on the request distribution of the clients. Moreover, server can dynamically allocate cache memory like the spider regulates tension on threads, based on the frequency of requests by each client. Also server can utilize the resources efficiently by providing more privileges to high requesting clients and utilize reclamation algorithms to clean up the cache memory. In this way, server memory load can be reduced and it can utilize resources efficiently.

Appendix

Codes

1. Compare the static memory allocation model with the dynamic memory allocation model

```
import random
def rmodel(server,client):
    d=[]
    equal=float(server)/client
    for index in xrange(client):
        request = random.randint(1,10)
        name="client %d"%(index+1)
        d+=[name,request,equal]
    return d

def smodel(server,client):
    sum=0
    d1=dict()
    final=[]
```

```
    for index in xrange(client):
        request=random.randint(1,10)
        # the times that one client request for the server
        name1="client %d"%(index+1)
        d1[name1]=request
        sum+=request
    distrit=float(server)/sum
    for index in xrange(client):
        name2="client %d"%(index+1)
        memory=round(float(distrit*d1[name2]))
        final+=[name2,d1[name2],memory]
    return final

def get2model():
    while True:
        try:
            server=int(raw_input("Enter the total memory of server: "))
            break
        except:
            print "Please enter a integer!"
    while True:
        try:
            client=int(raw_input("Enter How many clients: "))
            break
        except:
            print "Please enter a integer!"
    print "random model: "
    print rmodel(server,client)
    print "spider model: [client,request,server]"
    print smodel(server,client)
get2model()
```

2. Depicting the reclamation algorithm and the time difference in searching through the main memory and cache memory.

```
import random
import time
memory = set(range(1,10001))
cache = set()
mcache = []
seen = set()
seenAgain = set()
t = dict()
threshold = number**4/(memory**3/2*(10**8))
#algorithm to compute threshold
number=10000
#number of requests
for n in xrange(number):
    request = random.randint(1,10000)
    # check if the interval time is larger than the threshold
    start1 = time.time()
    if request in t:
```

```
        if request in cache:
            #print "time difference", start1 - t[request]
            if start1 - t[request] > threshold:
                #  print "wahoo"
                  cache.remove(request)
    t[request]=start1
    # search in cache memory
    start2 = time.time()
    if request in cache:
        end2= time.time()
        print ["cache","time = %0.30f milliseconds" % ((end2 - start2)*1000)]
    else:
        print "Not find request in cache memory!"
    # search in main memory
    start3 = time.time()
    if request in memory:
        end3 = time.time()
        print  ["memory","time = %0.30f milliseconds" % ((end3 - start3)*1000)]
    # check if there has request never been called when the length of cache is larger than 10
    # in cache memory. Then delete the the reuqest.
    if len(cache)<1000 :
        mcache += [request]
        if request not in cache:
            cache.add(request)
        cache1=cache.copy()
    else:
        for element in mcache:
            if (element in seen):
                seenAgain.add(element)
            seen.add(element)
        repeat = sorted(seenAgain)
        for e in cache1:
            if e not in repeat:
                cache.remove(e)
```

References

1. Kaihui Li; Yuanhai Zhang; Jin Xu; Changqiao Xu, "Dynamic Memory Allocation and Data Sharing Schedule in Media Server," Multimedia and Expo, 2007 IEEE International Conference on , vol., no., pp.72,75, 2-5 July 2007
2. Kensuke Nakata. Spatial learning affects thread tension control in orb-web spiders. Biology Letters, 9(4), 2013.
3. Jacquelyn M Zevenbergen, Nicole K Schneider, and Todd A Blackledge. Fine dining or fortress? Functional shifts in spider web architecture by the western black widow Latrodectus hesperus. Animal Behaviour, 76(3):823–829, 2008.