Visualising the Influence of Data Structure Choice on the Performance of a Distributed Database System.

Clare Churcher[1], Alan McKinnon[1], and Roger Jarquin[2]
[1]*Applied Computing, Mathematics and Statistics Group*
Lincoln University
P.O. Box 84, Lincoln University
Canterbury
New Zealand
email: churchec/mckinnon@lincoln.ac.nz

[2]*Aoraki Corporation*
P.O. Box 20152 Christchurch,
New Zealand
email: rjarquin@jade.co.nz

Abstract

The choice of data structure is an important decision in any software project. Application developers do not necessarily have a good understanding of how a particular system manages its data structures and how this might influence performance. In this paper we provide visualisations representing the behaviour of different data structures in an object oriented distributed database system. The data is obtained from cache monitoring software and the visualisations therefore represent the actual, as opposed to the theoretical, behaviour. The visualisations can be used as a supplement to the textual description of how a particular system manages its data structures so providing developers, educators and students with a clearer understanding of the implications of their choice of data structure.

1. Introduction

Any software application that manages data will use a variety of data structures. The comparative advantages and disadvantages of using B-Trees, linked lists or arrays are well understood at a theoretical level [1]. Depending on the type of application environment, the developer will have different levels of control over which data structures are used. In a relational database environment the developer will usually put data in high level relations and leave the low level handling of the data and indexes to the system. In other environments, the developer will have more control over the choice of data structures. An Object Oriented database such as JADE™ [2] or Jasmine™ [3] is likely to have a number of collection classes such as arrays, sets, lists, and B-Trees (keyed collections) for the developer to consider [4]. The developer's choice will be more informed if he or she has a clear understanding of how the different data structures perform in the system being used.

The implementations of data structures in different systems may affect the actual performance significantly. Also, generically named structures, such as arrays or collections, may be implemented quite differently in different systems. Most systems will provide documentation as to how the structures are implemented, but a busy developer with a deadline may consign this reading until "tomorrow". Even if the design and implementation of the structures is well understood it can still be difficult to anticipate what this actually means in practical terms.

In a distributed system the situation becomes more complex. A client may need to retrieve an object from a central server. There may be local caching on the client machine and there may also be locking involved, especially when objects are updated. Having a clear understanding of how arrays, lists or B-Trees operate in this more complex environment is not a trivial task.

In this paper we present some visualisations of the behaviour of different data structures in the JADE [2] object-oriented distributed processing system. For application developers and students it is hoped that the

1530-1362/00 $10.00 © 2000 IEEE

graphical representations will be more immediately understandable and create a more lasting impression than a solely textual description.

The visualisations in this paper are based on data collected from cache monitoring software available in the JADE kernel release 5.0. In this way we can illustrate how the different data structures actually behave. We can see which objects are being requested from the server, the locking that takes place, the make up of the cache, and the time involved. Pictures such as those provided here will be useful to developers, students and educators in understanding the impact of their choice of data structure. While the visualisations are based on data from a JADE system, the philosophy of presenting this type of information visually is quite general.

2. The Data

JADE is essentially a client-server system although other deployment options are possible [2]. The basic unit of information in the client cache managed by JADE is an object instance. These are ordered with the most recently used at the top of the cache and the least recently used (LRU) at the bottom. The latter is the next candidate for swapping out should space be needed for a new object.

Recently facilities have been incorporated into the JADE kernel to enable information about the current contents of the cache and about requests to the server to be captured from anywhere on the network. The information available with the requests includes:

- time (measured in Node Ticks each about 1 ms).
- type of request (e.g. get object, lock object, swap object out, put object to server)
- result of request (e.g. cache object updated by lock request)
- size of the object involved
- ID of the object involved
- duration of request (i.e. how long it took to go to the server and back)

Snapshots of the cache give information about each object currently in the cache. This information includes:

- the LRU order of the objects
- the ID of each object
- how long an object has been in the cache
- the number of operations on an object during its time in the cache

It is possible to produce a number of pictures using these variables. The visualisations in this paper compare 3 different variables on 2-D scatter plots with one variable along each axis and the points being coloured by the third. Many other possibilities exist but we have found that even these very simple pictures contain a considerable amount of information.

3. Test Data

A typical application involves the processing of large numbers of objects of the same type (e.g. customer). JADE has a number of different data structures for maintaining such collections of objects. A developer would often choose to maintain collections of objects in a MemberKeyDict. This is implemented as a B-Tree [1] and allows keyed access to each object. The structure consists of a collection header and then a tree of collection blocks. Each block contains the values of the keys and the addresses of the "next" collection blocks for a number of objects or references to the objects themselves. As we shall see, the number of objects that can be referenced by a single collection block depends on the size of the key. JADE has a Set data structure which is also implemented as a B-Tree but does not allow keyed access. In this paper we will use the MemberKeyDict as an example of the behaviour of B-Trees.

The array structure in JADE allows each element to be retrieved by its index in the array e.g. customerArray[n]. Currently this structure is implemented as a list. The list consists of a header and a list of collection blocks each with a number of references to other blocks or to elements that may be objects or primitive types. This data structure is not suitable for fast random access but adding objects can be extremely fast. We use the Array to illustrate the difference in cache activity that results from using a list data structure instead of a B-Tree structure.

To produce the data for the visualisations, we defined a customer class with four different sized attributes: An ID (integer), a short name (50 characters), a long name (100 characters) and a reference to a large bitmap object (900 KB) typical of a picture. We set up two B-tree structures (MemberKeyDicts) referencing the customer objects, one indexed by short name and the other by long name We also set up a list (Array) containing strings of short names.

We carried out a number of simple experiments such as traversing the objects in the structure. We ensured each object referenced was brought into the cache by accessing one of its attributes. If the attribute we accessed was the large bit map then the bit map (another object) also was brought into the cache which then filled quickly. We could therefore observe the behaviour of the structures when the cache was full.

The data produced by these experiments was stored in a file and analysed off-line. The recently released version 5.1 of JADE allows data to be obtained in real time.

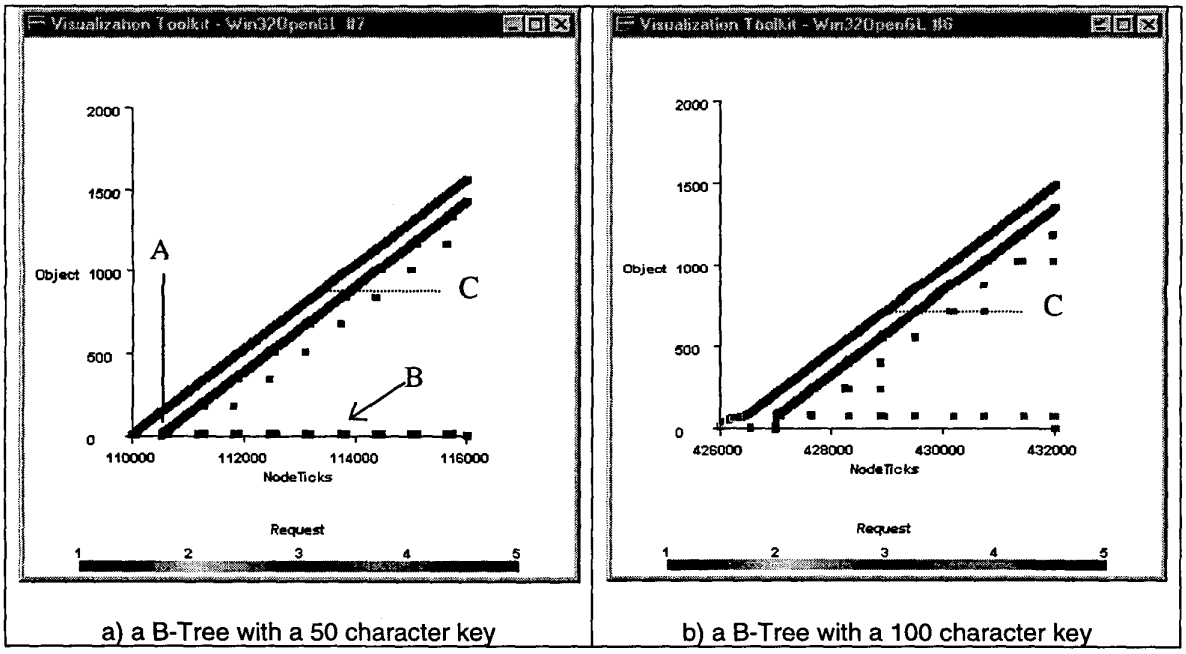
4. B-Trees structures.

When using the MemberKeyDict it is interesting to investigate how the size of the key affects the traversal of the collection of objects, especially when it is larger than the cache.

The diagrams in Figure 1 show the remote calls involved with the traversal of 800 customer objects using two MemberKeyDict collections, one with a 50 character key and one with a 100 character key. To do this each customer object is retrieved from the server and stored in the local cache. Each customer object contains a reference to a large bit map object. If we specifically access these objects for each customer they will also be retrieved from the server and the cache will fill very quickly. In this case objects at the bottom of the LRU order will be successively swapped out of the cache in order to make room for the next customer object.

The vertical axes in Figure 1 represent each object for which there is a remote call, numbered in the order they were first encountered. Time is represented by the horizontal axis. The points are coloured according to the type of request. In this case the light points (RequestType 1) represent objects coming into the cache, while the dark points (RequestType 5) indicate an object is being swapped out.

The line at (A) indicates where the cache is first filled and the first swap takes place. The two parallel lines are therefore showing the objects being read in and then swapped out a short time later. The customer objects being swapped out have no impact on performance as they are each only accessed once. However some collection headers and blocks are accessed more than once. The row of dark and light blocks along the bottom of Figure 1a (B) shows that the most frequently accessed collection block is being repeatedly swapped out only to be retrieved again almost immediately. This is also shown at (C) in both figures.



a) a B-Tree with a 50 character key

b) a B-Tree with a 100 character key

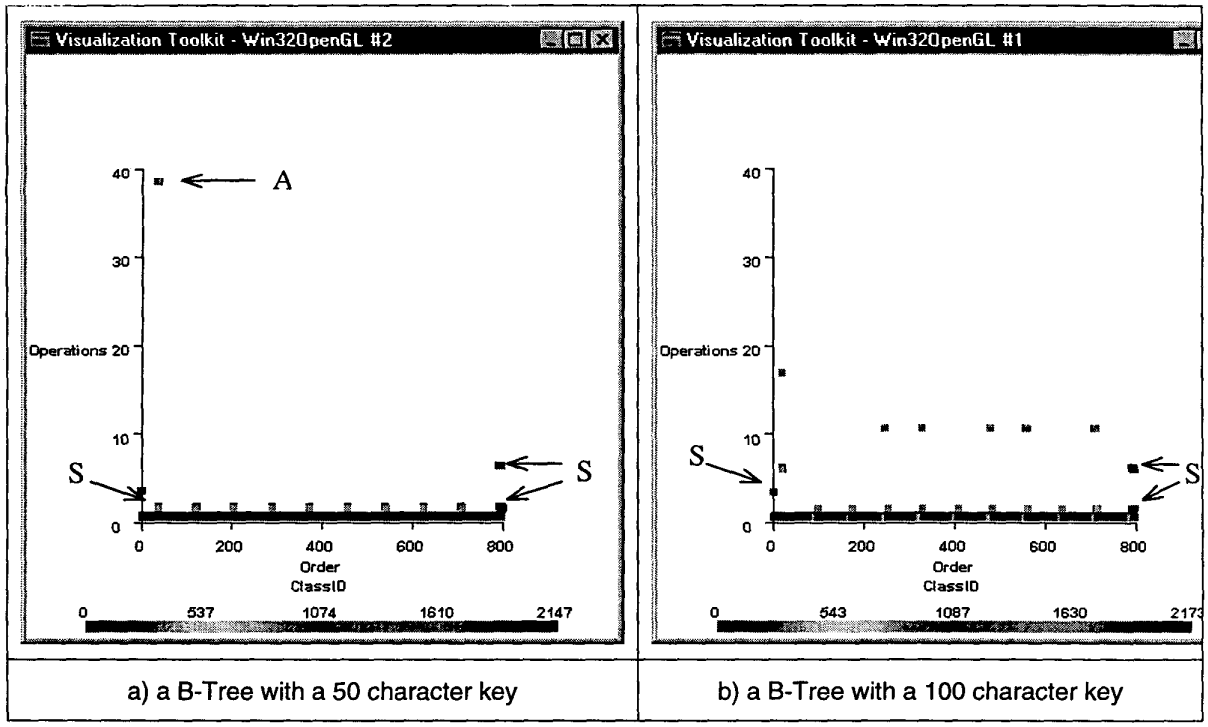
Figure 1: Traversing a B-Tree larger than the cache. The vertical axis represents each object for which there was a remote call, numbered in the order they were first encountered. The horizontal axis represents time. The objects are coloured according to the remote call (Request) type. Note that in b) there are some residual objects in the cache at the beginning of the traversal.

To understand the affect of the key size on the number of times a collection block is being recalled, it is useful to look at a snapshot of the cache at the end of the traversal. In this experiment we did not access the large bit map object so all the customer objects and collection headers were in the cache at the time the snapshots were taken. In Figure 2 each square is an object in the cache and is coloured by the ClassID. The dark squares are the customer objects while the lighter ones are the collection blocks. Along the bottom of each picture are the objects that are accessed just once. These include all the customer objects and also a number of leaf node collection block objects. In JADE the size of the collection blocks grows as more objects are added to the collection but there is a maximum size. With the 50 character key each leaf collection block can hold keys and references for about 20 objects so 40 blocks are required to keep track of the 800 objects. In Figure 2a we see one collection block (A) that is being accessed about forty times. We assume this is the next tier up in the B-Tree and is being accessed in order to reach each of the leaf nodes. It is this object that is being repeatedly swapped out and read in Figure 1a. Increasing the size of the cache slightly would prevent this object reaching the bottom of the LRU before it was required again.

With the larger key in Figure 2b, each leaf node can hold keys and references for only 10 objects, requiring 80 leaf nodes for the 800 customer objects in the collection. The next tier in this B-Tree has 7 collection block objects being accessed on average about 12 times (the first two have 6 and 17 operations, respectively), which is consistent with the number of leaf nodes. These 7 objects are the ones being swapped out and read back (in some cases twice) in Figure 1b.

In Figure 2 we also see 9 collection blocks that are accessed twice. Figures 1 and 2 do not shed much light on why this should be the case, although presumably it is a consequence of the detailed mechanism of the tree traversal.

In all the experiments in this paper there are a few system objects in the cache that appear as objects in the pictures (S).



a) a B-Tree with a 50 character key

b) a B-Tree with a 100 character key

Figure 2: Snapshots showing the objects in the cache at the end of a collection traversal. The vertical axis is the number of operations on each object and the horizontal axis is the position the object has ended up in the cache, 1 being the top of the LRU. System objects not directly related to the collection are indicated by S.

5. Traversing lists

In JADE, arrays are a specialisation of a generic list class. We use arrays to look at some of the implications of choosing a list structure.

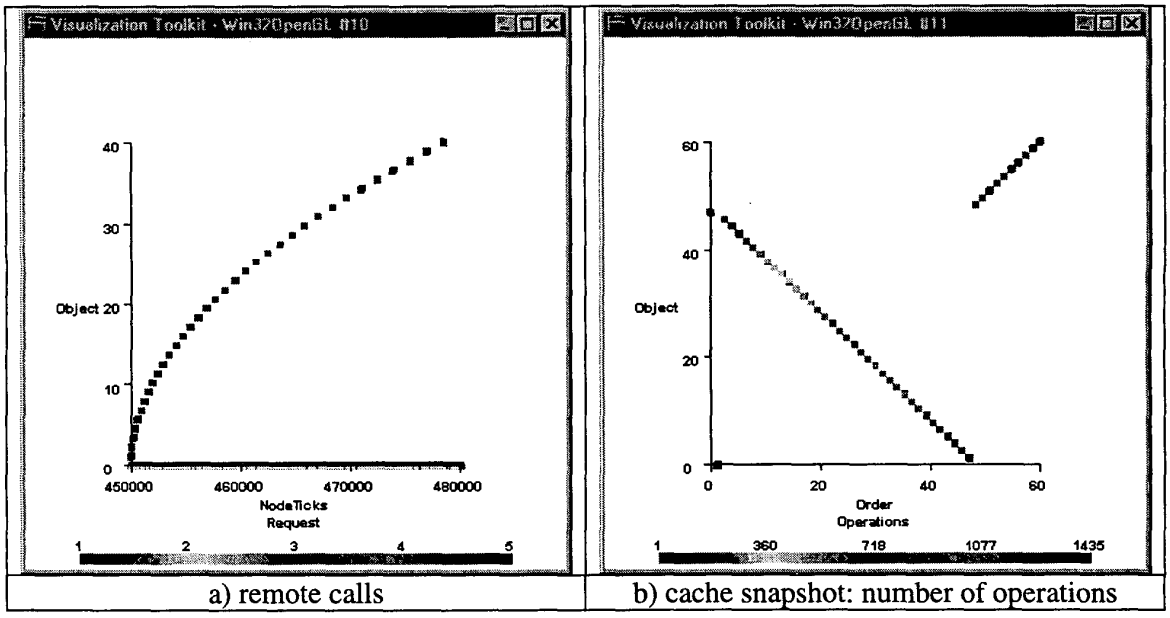
5.1. Traversing by index

To reference the ith element of an array requires all the previous elements to be accessed in turn. Figure 3 shows the consequences of accessing each element of the list of strings using the code below:

```
foreach index in 1 to 700 do
        stringvar := stringArray[index];
endforeach;
```

Figure 3a shows each of the remote calls to retrieve the array of strings into the cache. The objects being brought into the cache are the list nodes (collection blocks) which contain the strings. The cache never fills so no objects are swapped out. The time to access the later objects in the list increases as each of the increasing number of previous elements is traversed. Figure 3b shows each of the objects that is in the cache at the end of the traversal and is coloured by the number of operations. The elements at the beginning of the list have many operations (bottom of the long diagonal line) and this number of operations decreases until the last object which is accessed just the once. The short diagonal line in Figure 3b are system objects.

The horizontal line along the bottom of Figure3a shows a series of locks and unlocks. The header object of the list, is being locked each time an element of the list is accessed. JADE has a facility to prevent this repeated locking by using beginLoad/endLoad. A picture such as Figure3a sends a clear message to the developer or student that there is potential for improving performance.



a) remote calls

b) cache snapshot: number of operations

Figure 3: Traversing a List of Strings by Index. The vertical axis represents each object numbered in the order they were first encountered. The short diagonal line in b) are system objects present in the cache.

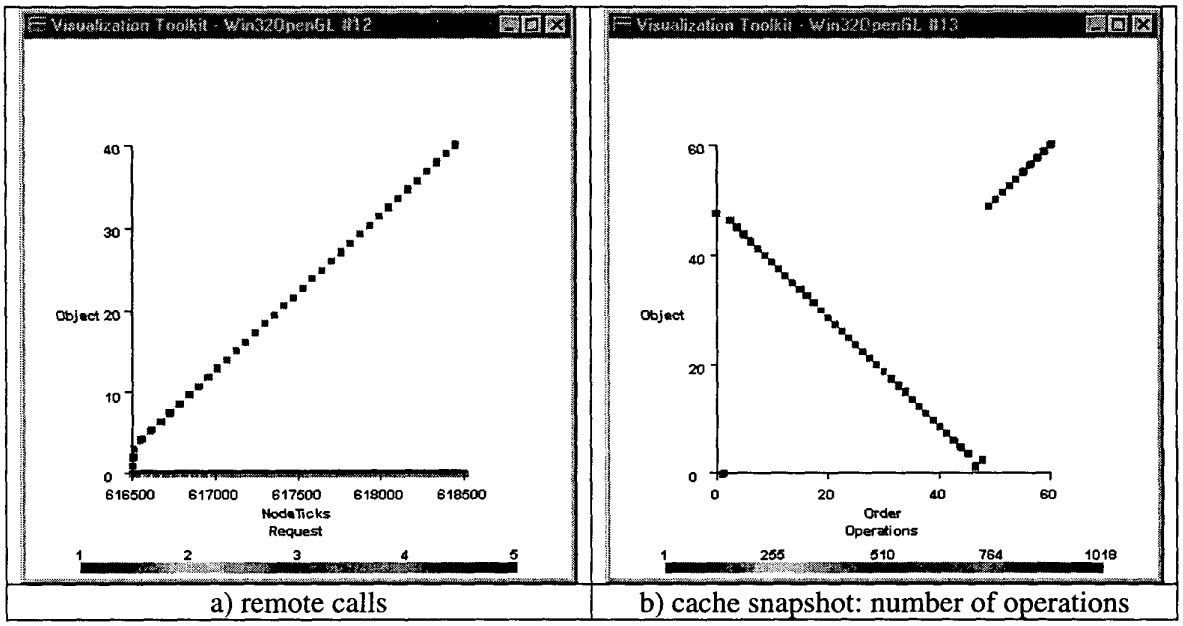
5.2. Traversing a List with an Iterator

Clearly traversing a list by accessing each element via its index is extremely inefficient. JADE has an iterator class which can be used to keep track of the current position in the list as it is traversed. The code segment below creates an iterator and uses it to traverse the list

```
iteratorObject:= ListObject.createIterator
while iteratorObject.next(stringvar) do
endwhile;
```

Figure 4b shows that each collection block object is now being accessed only once. A comparison of Figure3a and Figure 4a shows how much more efficient this method of traversal is especially towards the end of the List.

The pictures show real data of cache activities in each of these situations. While a developer or student may theoretically understand the concept of accessing a JADE array by index or by using an iterator, the pictures showing the actual performance will improve that understanding.



a) remote calls | b) cache snapshot: number of operations

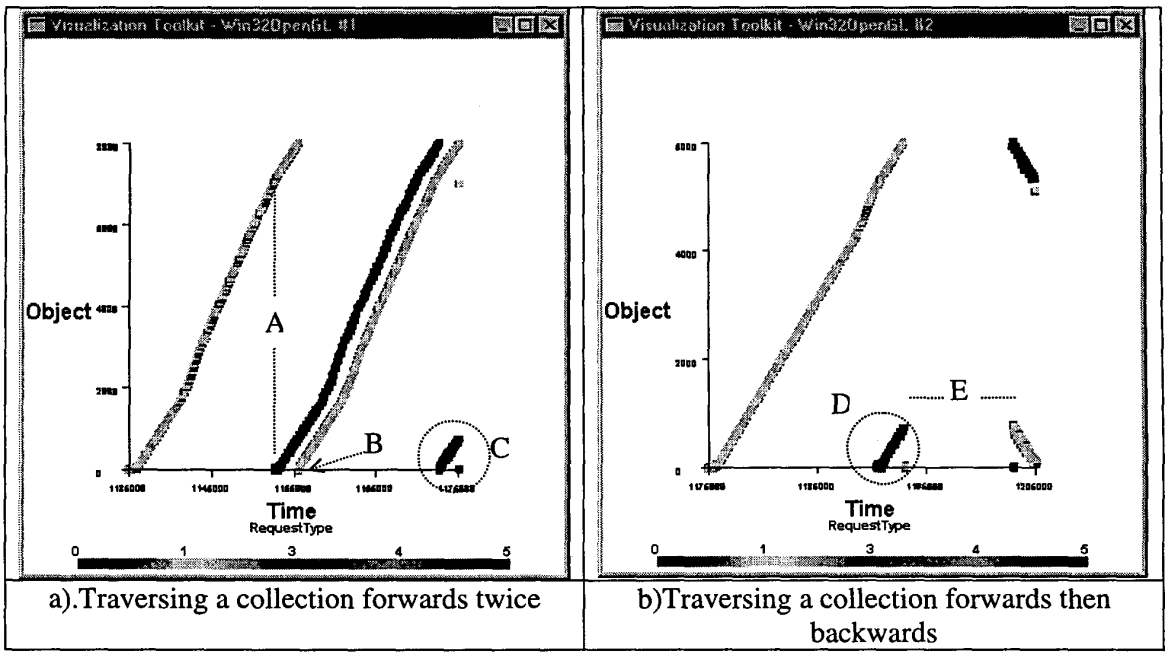
Figure 4: Traversing a List of Strings with an Iterator. The vertical axis represents each object numbered in the order they were first encountered. The short diagonal line in b) are system objects present in the cache.

6. Repeated Traversal of a Collection Larger than the Cache

Successive operations on a collection of objects is likely in a number of data processing situations. For example each object may need to be retrieved to calculate an average or total of some attribute and then the collection may be traversed again to compare each object with the result. If the total size of the objects is slightly greater than the cache then every object may be swapped out and then retrieved later.

Figure 5a show this situation for traversing a B-Tree twice.

The first light diagonal line in Figure 5a represents the objects in the collection being read into the cache for the first time. At (A) the first items in the collection begin to be swapped to make room for the last few objects. When the collection is traversed a second time (starting at B) the later objects are successively swapped out to allow for the earlier ones to be reloaded. The small horizontal distance between the dark diagonal line and the next light line indicates that the objects are being swapped out only to be reloaded a very short time later. The final few swaps (C) give an indication of how many objects in the collection cannot be accommodated by the cache – in this case about 10%.



a).Traversing a collection forwards twice

b)Traversing a collection forwards then backwards

Figure 5: Traversing a collection of objects slightly bigger than the cache.

If the developer sees this pattern (closely parallel dark and light lines) in a larger application he will be alerted to the fact that he is dealing with a collection slightly larger than the cache size and so may choose to adjust the size of the cache accordingly.

Another way to decrease the number of reloads when traversing a large collection twice is to make the second pass in reverse order. Figure 5b shows this case. While the first few objects (D) still need to be swapped out to accommodate the later objects, when the second pass is started in the reverse order the objects initially required are already in the cache. There are no remote calls nor corresponding network traffic during the time interval (E) until the traversal needs to reload the objects from the beginning of the collection. The negative gradient of the lines from this point reflects the fact that the objects are being retrieved in the reverse order from which we first encountered them.

7. Discussion

The behaviour of the cache in a distributed system such as JADE is inevitably complex. However, it is important for the application developer to have at least a conceptual understanding of how it operates so that applications are as efficient as possible. The developer must make decisions about the choice of data structure, such as arrays or indexed collections and also about how these data structures will be accessed during processing.

Software visualisation is intended to facilitate both the understanding and effective use of computer software [5]. We believe the visualisations shown in this paper are a useful complement to a textual description of how the cache behaves and would significantly enhance an application developer's understanding. They show the implications of both the choice of data structure and the processing done on it.

In this paper we have chosen to visualise the cache performance information at a level which shows the movement of objects and collection blocks into and out of the cache. We believe that is the level at which the application developer needs to have a conceptual understanding of how the system operates.

We could have chosen to visualise the cache performance information at a different level. For example, it would have been possible to explore in more detail how the system handles the collection headers and blocks that implement the B-Tree structure of the keyed collections.

This would be of interest to a person who is curious about the underlying structure of the system, but is not important knowledge for a developer. On the other hand, we have done some work that shows the developer how the cache management of the application is performing at a more macroscopic level by displaying histograms of different types of requests. This would be useful for a developer wishing to identify areas where the application is performing badly but does not add to understanding without a more detailed investigation.

The work reported here is part of a project to develop visualisations of the performance and behaviour of distributed systems such as JADE at levels that are appropriate to the various interests of the people who are involved with the software, be they the developers of the system itself or those that use it for developing applications.

8. References

[1] Model, Mitchell L. *Data Structures, Data Abstraction, a Contemporary Introduction Using C++* Prentice Hall NJ 1994.

[2] Aoraki Corporation *JADE* http://www.discoverjade.com cited March 2000.

[3] Computer associates International *Jasmine* http://www.cai.com cited May 2000

[4] Cattell, Rick Morgan Kaufman; *The Object Database Standard: ODMG 2.0*; 1997

[5] Price B.A., Baecker, R.M., and Small I.S. *A Principled Taxonomy of Software Visualization*. Journal of Visual Languages and Computing 4(3):211-266; 1993.