Locality of Reference

Intent

Improve application efficiency by exploiting the principle of Locality of Reference.

Also Known As

Caching

Motivation

Many applications continually reference large amounts of data. Often, the link to this data is slow, as in the cases of primary or secondary memory references, database queries, or network requests. This leads to poor performance in the application.

A common trait of most such applications is Locality of Reference. This principle observes that an application does not access all of its data at once with equal probability. Instead, it accesses only a small portion of it at any given time. An application can exhibit temporal and/or spatial locality. If some data is referenced, then there is a high probability that it will be referenced again in the near future. This is called temporal locality. If some data is referenced, then there is a high probability that data nearby will be referenced in the near future. This is called spatial locality.

An application can take advantage of Locality by keeping copies of the most often or more recently used data in a faster medium. This scheme is commonly known as "caching".

Take the example of an operating system. Ideally, we would like an unlimited amount of main memory, instantly accessible. In practice, we have a limited amount of main memory, and because it is cheaper, a very large amount of secondary memory. However the trade-off is that secondary memory tends to be several orders of magnitude slower than primary memory. We can approach the ideal by keeping the more often used data in main memory, and everything else in secondary memory. Because of the principle of Locality of Reference, we can be sure that most memory references will be to locations already stored in main memory, thereby improving efficiency and providing a flat memory model. This scheme is used in modern operating systems and is called virtual memory. Virtual memory gives users the appearance of unlimited primary memory by transparently utilizing secondary memory.

Applicability

Use the Locality of Reference pattern when:

- you have a data source hierarchy in which accessing different levels is orders of magnitude more expensive.
- data set is large.
- the probability of accessing any two members of the data set is unequal.
- this probability tends to change differentially.
- you have a single requester with a single cache (see **Consequences**).

When creating a caching system, policies for the following situations should be selected:

- where to place data in cache.
- how to find data if it is in cache.
- which data should be replaced on a cache miss.
- what occurs on a write miss.

Structure

```
class DataSource
{
    public:

        // Return the data stored at the address specified by Addr
        Data    read(Addr);

        // Write Data to location Addr
        void    write(Addr, Data);

    private:

        // The raw data.  Can be stored in any data structure, though this should be chosen
        // with efficiency in mind.
        Data    theData;
}
```

```
class Cache
{
    public:

        // Insert Data located at Addr into cache.  If the cache is full, return False.
        bool        insert(Addr, Data);

        // Remove data associated with address Addr from the cache
        void        remove(Addr);

    private:

        Data        theDataSubset;

        // Returns true if the requested data is in cache
        bool        isInCache(Addr);

}

class CacheManager
{
    public:

        Data        read(Addr);
        void        write(Addr, Data);

    private:

        // Suggests an entry in the cache to remove in order to make room for new entry
        Addr        replacementAlgorithm(void);

        // References to the data and caching stores
        DataSource*     dataSourceRef;
        Cache*          cacheRef;
}
```

Participants

The following are the actors involved in implementing a caching scheme.

Data Source

This is the original data source. All data resides here and may be expensive to access.

Data Requester

This is an application which has requested data which is in the Data Source.

Cache

The Cache is a small storage area which keeps a subset of the Data Source. Accessing data stored in the Cache is faster than accessing it from the original Data Source.

Cache Manager

The Cache Manager is the Data Requester's interface to the Data Source. It handles all data requests, looking first in the Cache, and then in the Data Source if not found.

The Cache Manager also has the responsibility of initially populating the Cache, and replacing entries when it becomes full.

The Cache Manager can be considered an instance of the Proxy pattern [1]. It shields the requesters from process of searching both the Cache and the Data Source.

Collaborations

There are two operations one can perform on the Data Source. The first is to read data, and the second is to write data. This section describes how the Data Source, the Data Requester, the Cache, and the Cache Manager work together in various scenarios.

Read requests

On a read request, the Cache Manager searches for the data in the Cache. If the data is found, it is returned to the requester. This is called a "hit."

If the data was not in the Cache, this is called a "miss." In this case the manager gets the data from the Data Source and returns it to the requester. In addition, the Cache Manager inserts this data into the cache so that future requests will not require a trip to the Data Source. If the cache is full, the manager must make room for the new entry by removing an existing one.

Write requests

On a write request, the Cache Manager can act differently for write hits and write misses, depending on which policies are chosen.

Write Hits

The policy of a write to the Cache always resulting in a write directly to the Data Source is called "write-through."

The "write-back" policy only writes to the Data Source when that cache entry is about to be removed (to make room for another entry).

Write Misses

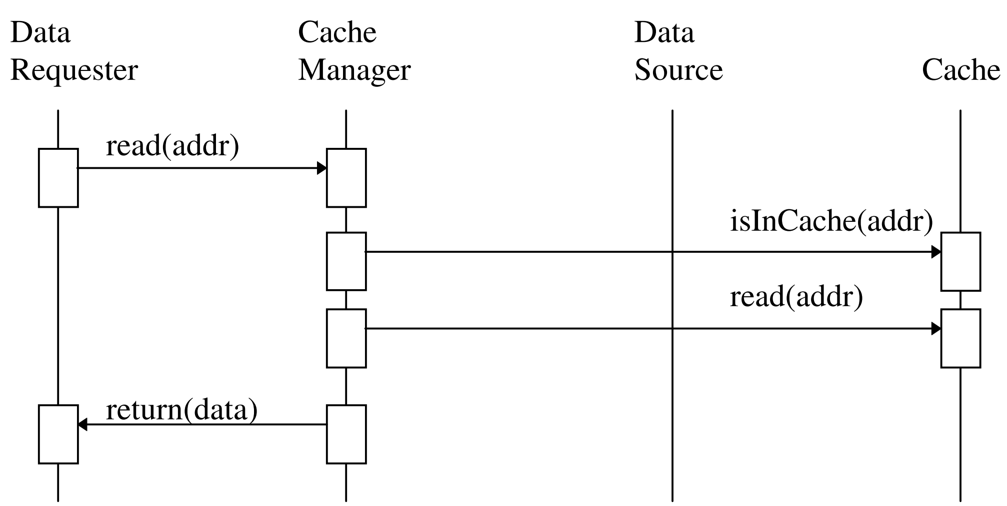
If the data is not in the Cache, one of two policies can be used: "fetch-on-write" and "write-around."   Both policies immediately update the Data Source. However, fetch-on-write also places the last request into the Cache. The cost of the additional operation can be recovered if there are subsequent read requests for that data.

Cache Insertion and Removal Policies

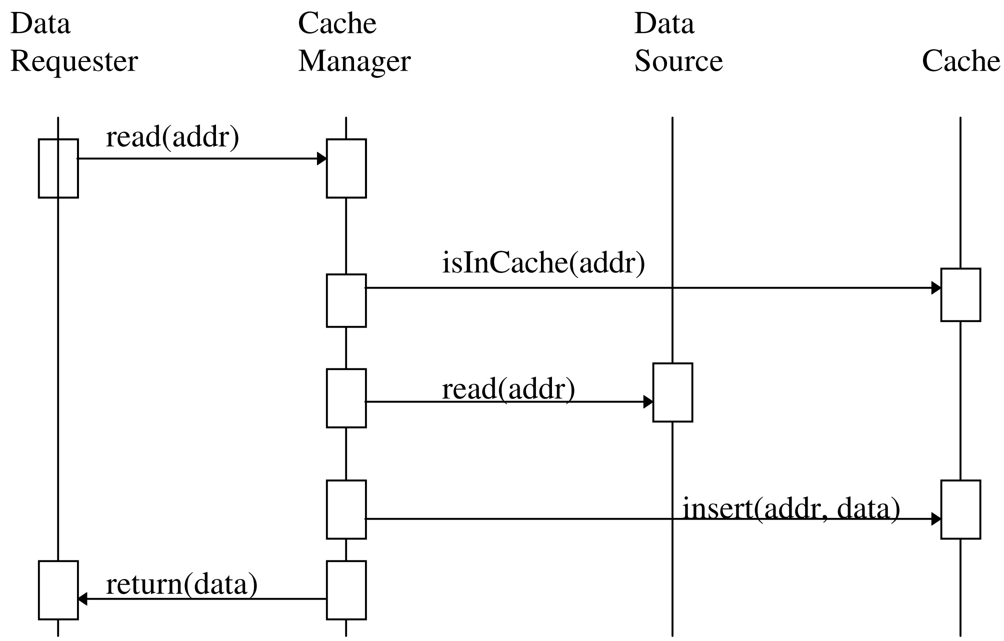
When the cache is full, the Cache Manager must make room by removing an existing cache entry. Two possibilities are to remove the least recently used or the least frequently used. Keeping the more recently used data in cache exploits temporal locality.

When inserting data into the cache, an application can take advantage of spatial locality by also inserting multiple data items that are logically or physically contiguous to the original request. For example, if someone requested data element number 1000, there is a reasonable chance that they will request data element number 1001 next.

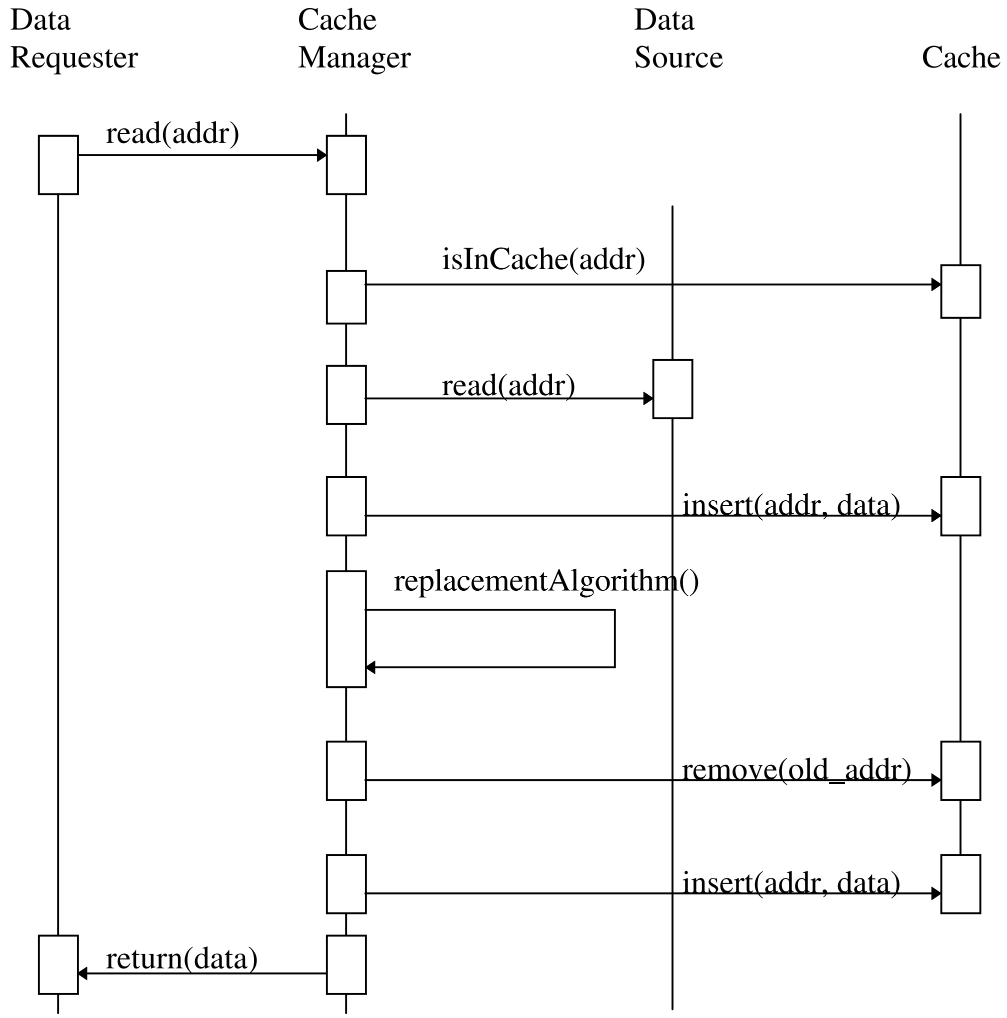
A "read-hit"



A “read-miss,” cache is not full



A "read-miss," cache is full



Consequences

Data integrity becomes difficult and sometimes impossible when you extend the problem to allow multiple Data Requesters, each with their own cache.

The Cache of every requester is in danger becoming stale whenever one requester performs a write. The Observer pattern can be used to notify all caches to invalidate their affected entries, but this adds a lot of overhead.

Known Uses

These principles have been used in the areas of computer architecture, operating system, and database design and implementation for over fifteen years.

Hardware

- Primary memory cache (or memory hierarchies)
- Associative memory, transaction lookaside buffer

Software

- Virtual memory paging algorithm
- Web servers
- Databases

Related Patterns

Observer [1]

Can be used to notify multiple Data Requesters of changes in data. A specific example of this is a scheme where multiple microprocessors share a single bus. Each of them polls the bus listening for write requests. When one comes along, the appropriate data is invalidated in each cache. This is known as "snooping."

Proxy Cache [3]

This is a related pattern, but it focuses on a different area. One way to look at it is that the Proxy Cache pattern is a Proxy which uses Locality of Reference to improve the speed of its transactions.

Acknowledgments

DeLoy Bitner, Brad Flood, Brett Moser, Len Myers, Doug Schmidt

References

[1] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[2] Coplien, James O., Douglas C. Schmidt. *Pattern Languages of Program Design.* Reading, MA: Addison-Wesley, 1995.

[3] Vlissides, John M., James O. Coplien, Norman L. Kerth. *Pattern Languages of Program Design 2.* Reading, MA: Addison-Wesley, 1996.

[4] Patterson, David A., John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface.* San Mateo, CA: Morgan Kaufmann, 1994.

[5] Patterson, David A., John L. Hennessy. *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 1996.