Locality of reference

From Wikipedia, the free encyclopedia

In computer science, locality of reference, also known as the principle of locality, is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern. There are two basic types of reference locality – temporal and spatial locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

Locality is merely one type of predictable behavior that occurs in computer systems. Systems that exhibit strong *locality of reference* are great candidates for performance optimization through the use of techniques such as the caching, prefetching for memory and advanced branch predictors at the pipelining stage of processor core.

Contents

Types of locality

There are several different types of locality of reference:

Temporal locality
: If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case it is common to make efforts to store a copy of the referenced data in special memory storage, which can be accessed faster. Temporal locality is a special case of spatial locality, namely when the prospective location is identical to the present location.

Spatial locality
: If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access.

Memory locality
: Spatial locality explicitly relating to memory.

Branch locality
: If there are only a few possible alternatives for the prospective part of the path in the *spatial-temporal coordinate space.* This is the case when an instruction loop has a simple structure, or the possible outcome of a small system of conditional branching instructions is restricted to a small set of possibilities. Branch locality is typically not a spatial locality since the few possibilities can be located far away from each other.

Equidistant locality
: It is halfway between the spatial locality and the branch locality. Consider a loop accessing locations in an equidistant pattern, i.e. the path in the *spatial-temporal coordinate space* is a dotted line. In this case, a simple linear function can predict which location will be accessed in the near future.

In order to benefit from the very frequently occurring *temporal* and *spatial* kind of locality, most of the information storage systems are hierarchical; see below. The *equidistant* locality is usually supported by the diverse nontrivial increment instructions of the processors. For the case of *branch* locality, the contemporary processors have sophisticated branch predictors, and on the basis of this prediction the memory manager of the processor tries to collect and preprocess the data of the plausible alternatives.

Relevance for locality

There are several reasons for locality. These reasons are either goals to achieve or circumstances to accept, depending on the aspect. The reasons below are not disjoint; in fact, the list below goes from the most general case to special cases:

Predictability

In fact, locality is merely one type of predictable behavior in computer systems.

Structure of the program

Locality occurs often because of the way in which computer programs are created, for handling decidable problems. Generally, related data is stored in nearby locations in storage. One common pattern in computing involves the processing of several items, one at a time. This means that if a lot of processing is done, the single item will be accessed more than once, thus leading to temporal locality of reference. Furthermore, moving to the next item implies that the next item will be read, hence spatial locality of reference, since memory locations are typically read in batches.

Linear data structures

Locality often occurs because code contains loops that tend to reference arrays or other data structures by indices. Sequential locality, a special case of spatial locality, occurs when relevant data elements are arranged and accessed linearly. For example, the simple traversal of elements in a one-dimensional array, from the base address to the highest element would exploit the sequential locality of the array in memory.[1] The more general *equidistant locality* occurs when the linear traversal is over a longer area of adjacent data structures with identical structure and size, accessing mutually corresponding elements of each structure rather than each entire structure. This is the case when a matrix is represented as a sequential matrix of rows and the requirement is to access a single column of the matrix.

Efficiency of memory hierarchy use

Although random access memory presents the programmer with the ability to read or write anywhere at any time, in practice latency and throughput are affected by the efficiency of the cache, which is improved by increasing the locality of reference. Poor locality of reference results in cache thrashing and cache pollution and to avoid it, data-elements with poor locality can be bypassed from cache.[2]

General locality usage

If most of the time the substantial portion of the references aggregate into clusters, and if the shape of this system of clusters can be well predicted, then it can be used for performance optimization. There are several ways to benefit from locality using optimization techniques. Common techniques are:

Increasing the locality of references
This is achieved usually on the software side.

Exploiting the locality of references
This is achieved usually on the hardware side. The *temporal* and *spatial* locality can be capitalized by hierarchical storage hardwares. The *equidistant* locality can be used by the appropriately specialized instructions of the processors, this possibility is not only the responsibility of hardware, but the software as well, whether its structure is suitable for compiling a binary program that calls the specialized instructions in question. The *branch* locality is a more elaborate possibility, hence more developing effort is needed, but there is much larger reserve for future exploration in this kind of locality than in all the remaining ones.

Spatial and temporal locality usage

Hierarchical memory

Hierarchical memory is a hardware optimization that takes the benefits of spatial and temporal locality and can be used on several levels of the memory hierarchy. Paging obviously benefits from *temporal and spatial locality*. A cache is a simple example of exploiting temporal locality, because it is a specially designed, faster but smaller memory area, generally used to keep recently referenced data and data near recently referenced data, which can lead to potential performance increases.

Data-elements in a cache do not necessarily correspond to data-elements that are spatially close in the main memory; however, data elements are brought into cache one cache line at a time. This means that spatial locality is again important: if one element is referenced, a few neighboring elements will also be brought into cache. Finally, temporal locality plays a role on the lowest level, since results that are referenced very closely together can be kept in the machine registers. Some programming languages (such as C) allow the programmer to suggest that certain variables be kept in registers.

Data locality is a typical memory reference feature of regular programs (though many irregular memory access patterns exist). It makes the hierarchical memory layout profitable. In computers, memory is divided into a hierarchy in order to speed up data accesses. The lower levels of the memory hierarchy tend to be slower, but larger. Thus, a program will achieve greater performance if it uses memory while it is cached in the upper levels of the memory hierarchy and avoids bringing other data into the upper levels of the hierarchy that will displace data that will be used shortly in the future. This is an ideal, and sometimes cannot be achieved.

Typical memory hierarchy (access times and cache sizes are approximations of typical values used as of 2013 for the purpose of discussion; actual values and actual numbers of levels in the hierarchy vary):

- CPU registers (8-256 registers) – immediate access, with the speed of the inner most core of the processor
- L1 CPU caches (32 KiB to 512 KiB) – fast access, with the speed of the inner most memory bus owned exclusively by each core

- L2 CPU caches (128 KiB to 24 MiB) – slightly slower access, with the speed of the memory bus shared between twins of cores
- L3 CPU caches (2 MiB to 32 MiB) – even slower access, with the speed of the memory bus shared between even more cores of the same processor
- Main physical memory (RAM) (256 MiB to 64 GiB) – slow access, the speed of which is limited by the spatial distances and general hardware interfaces between the processor and the memory modules on the motherboard
- Disk (virtual memory, file system) (1 GiB to 256 TiB) – very slow, due to the narrower (in bit width), physically much longer data channel between the main board of the computer and the disk devices, and due to the extraneous software protocol needed on the top of the slow hardware interface
- Remote Memory (such as other computers or the Internet) (Practically unlimited) – speed varies from very slow to extremely slow

Modern machines tend to read blocks of lower memory into the next level of the memory hierarchy. If this displaces used memory, the operating system tries to predict which data will be accessed least (or latest) and move it down the memory hierarchy. Prediction algorithms tend to be simple to reduce hardware complexity, though they are becoming somewhat more complicated.

Matrix multiplication

A common example is matrix multiplication:

```
for i in 0..n
 for j in 0..m
  for k in 0..p
   C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

By switching the looping order for `j` and `k`, the speedup in large matrix multiplications becomes dramatic, at least for languages that put contiguous array elements in the last dimension. This will not change the mathematical result, but it improves efficiency. In this case, "large" means, approximately, more than 100,000 elements in each matrix, or enough addressable memory such that the matrices will not fit in L1 and L2 caches.

```
for i in 0..n
 for k in 0..p
  for j in 0..m
   C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The reason for this speed up is that in the first case, the reads of `A[i][k]` are in cache (since the `k` index is the contiguous, last dimension), but `B[k][j]` is not, so there is a cache miss penalty on `B[k][j]`. `C[i][j]` is irrelevant, because it can be factored out of the inner loop. In the second case, the reads and writes of `C[i][j]` are both in cache, the reads of `B[k][j]` are in cache, and the read of `A[i][k]` can be factored out of the inner loop. Thus, the second example has no cache miss penalty in the inner loop while the first example has a cache penalty.

On a year 2014 processor, the second case is approximately five times faster that the first case, when written in C and compiled with `gcc -O3`. (A careful examination of the disassembled code shows that in the first case, gcc uses SIMD instructions and in the second case it does not, but the cache penalty is much worse than the SIMD gain).

Temporal locality can also be improved in the above example by using a technique called *blocking*. The larger matrix can be divided into evenly sized sub-matrices, so that the smaller blocks can be referenced (multiplied) several times while in memory.

```
for (ii = 0; ii < SIZE; ii += BLOCK_SIZE)
 for (kk = 0; kk < SIZE; kk += BLOCK_SIZE)
  for (jj = 0; jj < SIZE; jj += BLOCK_SIZE)
   for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++)
    for (k = kk; k < kk + BLOCK_SIZE && k < SIZE; k++)
     for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The temporal locality of the above solution is provided because a block can be used several times before moving on, so that it is moved in and out of memory less often. Spatial locality is improved because elements with consecutive memory addresses tend to be pulled up the memory hierarchy together.

See also

- Burst mode (computing)
- Cache-oblivious algorithm
- File system fragmentation
- Partitioned global address space
- Row-major order
- Scalable locality
- Scratchpad memory
- Data oriented design
- memory access pattern
- Working set

Bibliography

- P.J. Denning, The Locality Principle, Communications of the ACM, Volume 48, Issue 7, (2005), Pages 19–24
- P.J. Denning, S.C. Schwartz, Properties of the working-set model, Communications of the ACM, Volume 15, Issue 3 (March 1972), Pages 191-198

References

1. Aho, Lam, Sethi, and Ullman. "Compilers: Principles, Techniques & Tools" 2nd ed. Pearson Education, Inc. 2007
2. "A Survey Of Cache Bypassing Techniques (https://www.academia.edu/24842555/A_Survey_of_Cache_Bypassing_Techniques)", JLPEA, vol. 6, no. 2, 2016

Retrieved from "https://en.wikipedia.org/w/index.php?title=Locality_of_reference&oldid=766670092"

Categories: Computer memory | Cache (computing) | Software optimization

---