On Demand Readahead

Readahead Algorithms

Principles of 2.6 readahead

Sequential detection

- 파일을 처음 읽을 때
- 이전 페이지에 연속된 페이지를 읽을 때

위의 상황이 아니면 random read 이다.

Readahead size

- Initial: current_window 나 ahead_window 가 없을 경우, 첫 readahead 의 크기는 현재 읽기 요청의 사이즈로 추론된다. 일반적으로 readahead_size 는 read_size 의 2~4 배가 된다.
- Ramp-up: 이전 readahead 가 있을 경우, 크기는 2~4 배가 된다.
- Full-up: max_readahead 에 도달할 경우를 의미

요청 크기가 충분히 클 경우(10MB), readahead 크기는 바로 full-up 이 된다.

Readahead pipelining

Processing 시간과 디스크의 I/O 시간을 최대한 overlap 하기 위해서 2 개의 readahead window 를 유지한다.

- current_window: 어플리케이션이 작동할 곳
- ahead_window: 비동기 I/O 가 발생할 때, 요청이 sequential request (oversize, current_window 만 있을 경우, crossed into ahead_window) 일 때, ahead_window 가 미리 열리고, 새롭게 생긴다.

Cache hit/miss

- readahead cache hit: readahead 되는 Page 가 이미 캐시되어 있는 경우, readahead cache hit 이 오래 발생된 경우에는 이미 file 이 cache 되어 있는 경우를 의미한다. Threshold 가 VM_MAX_CACHE_HIT(= 256)이 될 경우, readahead 는 필요없는 Page-cache 를 찾는 과정을 피하기 위해서 꺼진다.

- readahead cache miss: read 할 때 이미 readahead 로 가져온 page 가 없는 경우, readahead size 가 너무 큰 경우를 의미하고 다음 readahead 크기를 2 로 줄인다.

```
do_generic_mapping_read:
    call page_cache_readahead
    for each page
        if is prev_page + 1
            call page_cache_readahead
        if page not cached
            report cache miss
            leave cache hit mode

page_cache_readahead:
    handle unaligned read
    set prev_page to current page index
    if in cache hit mode
        return
    shift prev_page to the last requested page,
        but no more than max_readahead pages
    if is sequential read and no current_window
        make current_window
        call blockable_page_cache_readahead
        if is oversize read
            call make_ahead_window
    elif is random read
        clear readahead windows
        limit size to max_readahead
        call blockable_page_cache_readahead
    elif no ahead_window
        call make_ahead_window
    elif read request crossed into ahead_window
        advance current_window to ahead_window
        call make_ahead_window
    ensure prev_page do not overrun ahead_window

make_ahead_window:
    if have seen cache miss
        clear cache miss status
        decrease readahead size by 2
    else
        x4 or x2 readahead size
        limit size to max_readahead
    call blockable_page_cache_readahead

blockable_page_cache_readahead:
    if is blockable and queue congested
        return
    submit readahead io
    if too many continuous cache hits
        clear readahead windows
        enter cache hit mode
```

Figure 1: readahead in 2.6.20

On-demand Algorithm

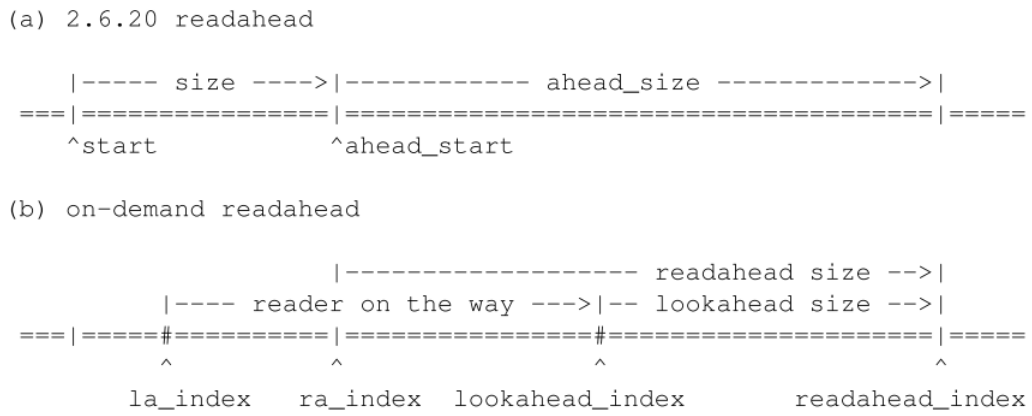

Figure 3: tracking readahead windows

Readahead windows

Readahead pipelining 를 위해서 두개의 window 를 사용했다. Application 이 current_window 에서 작동하고 있을 때, I/O 는 ahead_window 에서 작동한다. 알고리즘 구현이 쉽지 않다. 두개의 window 를 유지하는 이유는 파이프라이닝이다. Sequential readahead 에 필요한 기능을 파악해서 간단하게 구현하는 것이 목표이다.

파이프라이닝을 하기 위해서 lookahead size 라는 threshold 를 두고, 다음 readahead 를 진행하기 위해서, 아직 소모되지 않은 readahead page 의 수가 lookahead size 밑으로 감소되기 전에 I/O 를 발생시킨다. (lookahead size = 0 이면 파이프라이닝을 사용하지 않고, lookahead size 가 readahead size 와 같은 경우에는 파이프라이닝을 최대한 사용)

Readahead on demand

Linux 2.6 readahead 는 모든 read request 를 조사해서, 연속 접근인지 파악했다. 과도한 page_cache_readahead() call 을 발생, convoluted feedback loops 에서 cache hits/misses 를 다뤄야하는 문제가 있다.

Readahead 는 아래 두가지 경우에만 실행된다.

- Sync readahead on cache miss: cache miss 가 발생했을 때, 어플리케이션이 I/O 를 진행중인 상황에서 readahead 를 시도하고 더 많은 페이지를 읽어야하는지 체크
- Async readahead on lookahead page: readahead 크기가 lookahead 크기보다 작을 경우, readahead 를 해야한다.

The new algorithm (on-demand readahead)

대부분 기존 알고리즘과 비슷하게 작동하지만 아래와 같은 경우 다르다.

- 새로운 parameter page 가 ondeman_readahead()에 주어진다. 이것은 현재 페이지가 존재하고 있는지를 말해준다. NULL 이면 synchronous readahead, 그렇지않으면 asynchronous 를 의미한다.
- 현재 어디에서 읽기 요청이 시작되는지를 알려주는 begin offset 이 새롭게 생겼다. Prev_page 는 이전 요청의 마지막 접근된 페이지를 알려준다. 새로운 연속 접근은 begin offset – prev page <= 1 일 경우에 연속 접근이라고 판단한다.
- Overlapped random 읽기를 위한 I/O 가 이전에는 잘 지원되지 않았다. 8page random read 에서 처음 4 페이지는 이전 읽기에 overlapped 되어있다고 가정해보자. 2.6 readahead 는 첫 페이지 접근 전에 8 페이지 모두에 요청을 보낼 것이다. 하지만 새로운 on-demand readahead 는 5 번째 페이지에 접근을 할 경우 남은 4 개의 페이지에만 요청을 보낸다. 이는 불필요한 페이지 캐시를 lookup 을 피할 수 있다. ?
  (Hence it avoids some unnecessary page-cache lookups, at the cost of not being able to overlap transfer of the leading cached pages with I/O for the follow- ing ones.)
- 리눅스 2.6.20 은 연속 읽기 요청에 대해서 밖에 readahead 를 지원하지 않았다. 새로운 디자인에서는 lookahead hit 이 다음 readahead 를 발생시킬 수 있다. 이것은 interleaved read 에 대한 탐지를 가능하게 하였다.

```
do_generic_mapping_read:
    for each page
        if page not cached
            call ondemand_readahead
        if page has lookahead mark
            call ondemand_readahead
    set prev_page to last accessed page

ondemand_readahead:
    if is asynchronous readahead and queue congested
        return
    if at start of file
        set initial sizes
    elif is small random read in wild
        read as is
        return
    elif at lookahead_index or readahead_index
        ramp up sizes
    else
        set initial sizes
        if has lookahead mark
            ramp up size
    fill readahead state
    submit readahead io
    set lookahead mark on the new page at new lookahead_index
```

Figure 4: on-demand readahead algorithm

Benchmarks

1. sequential re-read in 4KB, sequential re-read in 1MB
    a. no readahead invocation 으로 인한 성능향상
2. small file re-read (tar /lib)
    a. no page-cache lookup 으로 인한 성능향상
3. random reading sparse file
    a. one extra page-cache lookup per cache miss 로 성능 하락
4. sequential reading sparse file
    a. less readahead invocation 으로 인항 성능 향상
5. iozone -c -t1 -s 4096m -r 64k (64KB non-overlapping reads on a 4GB file)