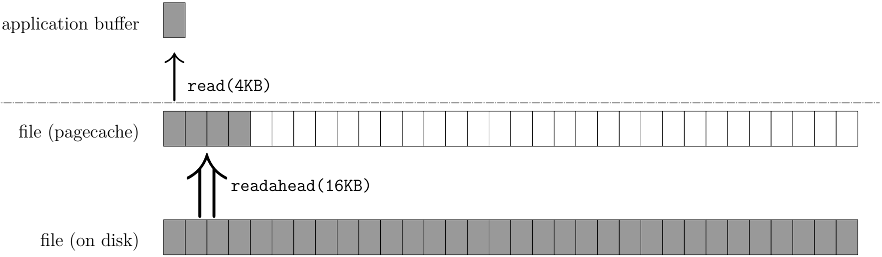
Readahead Algorithms and I/O Performance

Fengguang Wu

2008-2-19

About readahead

- for better I/O performance
  - higher throughput: small reads => large readahead I/O
  - lower latency: sync reads => async readahead I/O
- hard to get right
  - all kinds of I/O access patterns
  - pagecache dynamics (cache hits, readahead thrashing)



Sequential reads

- `sequential = (index == prev_index + 1);`

1-page reads

```
t0  +
t1   +
t2    +
t3     +
t4      +
t5       +
t6        +
t7         +
t8          +
t9           +
```

2-page reads

```
t0  ++
t1    ++
t2      ++
t3        ++
t4          ++
t5            ++
t6              ++
t7                ++
t8                  ++
t9                    ++
```

Unaligned reads

- read requests not aligned to 4KB page boundaries

-
  ```
  sequential = (index == prev_index  ||
                index == prev_index + 1);
  ```

1KB reads

```
t0  +
t1  +
t2  +
t3  +
t4   +
t5   +
t6   +
t7   +
t8    +
t9    +
```

4KB reads

```
t0  ++
t1   ++
t2    ++
t3     ++
t4      ++
t5       ++
t6        ++
t7         ++
t8          ++
t9           ++
```

10KB reads

```
t0 +++
t1   +++
t2      +++
t3        +++
t4           +++
t5             +++
t6                +++
t7                  +++
t8                     +++
t9                       +++
```

Retried reads

- pages transfered < pages requested
- occur in
  - retry based AIO
  - non-blocking IO

requested pages (2.6.22)

```
t0  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
t1                  ++++++++++++++++++++++++++++++++++++++++++++
t2                                  ++++++++++++++++++++++++++++
t3                                                  ++++++++++++
```

transfered pages (2.6.23)

```
t0  ----------------
t1                  ----------------
t2                                  ----------------
t3                                                  ------------
```

Retried reads performance

- workload
    - AMD Opteron 250 server with 16G mem
    - lighttpd serving 1200 clients
- performance

	vanilla	context	
avg readahead size(pages)	15	228	x15
cpu %iowait	25.20	21.00	-16.7%
disk %util	13.03	9.62	-26.2%
network bandwidth(MB/s)	37.00	43.40	+17.3%
disk bandwidth(MB/s)	28.18	36.46	+29.4%

- another lighttpd user
    - ‘IO-Wait has dropped significantly from 80% to 20%.’

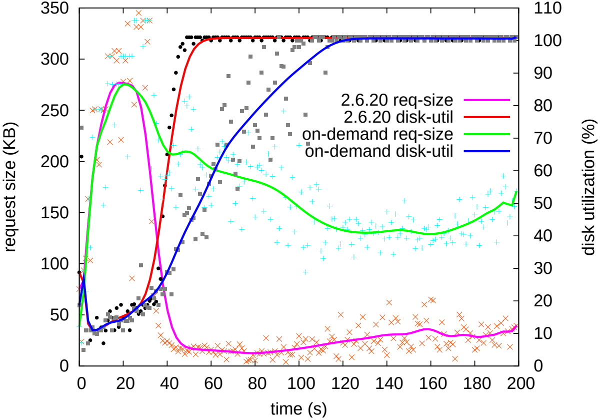
Readahead thrashing

- readahead pages drop out from LRU cache before being visited

```
t0  ------*____________
t1  -------*___________
t2  --------*__________
    (readahead thrashing)

t3           *             2.6.22:
t4           -*              pages are faulted in one by one
t5           --*
t6           ---*

t3           *___          2.6.23:
t4           -*__________    readahead window re-established
t5           --*_________
t6           ---*________

    [-] visited page  [*] reader position  [_] readahead page
```

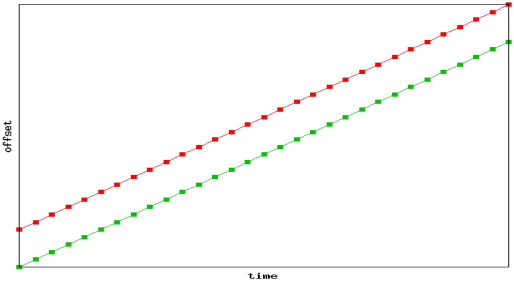
Readahead thrashing performance

- workload
  - 128MB memory, 1MB max readahead
  - starts one new 100KB/s stream for every second
- performance
  - max network throughput: $5MB/s => 15MB/s$
  - min I/O size: $5KB => 40KB$

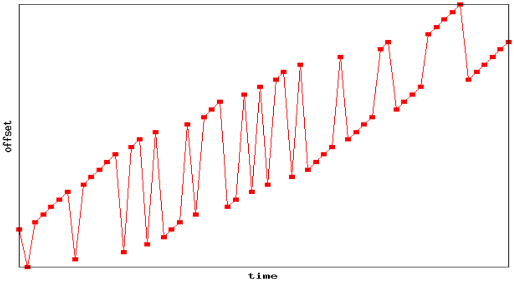


Interleaved reads

- 2 streams on 2 fds



- 2 streams on 1 fd



Interleaved reads: data structures

Linux: 1 stream per fd

```
struct inode --+--- struct file --- struct file_ra_state -- prev_index
               |
               +--- struct file --- struct file_ra_state -- prev_index
               |
               +--- struct file --- struct file_ra_state -- prev_index
```

Solaris/ZFS: many streams per file

```
struct dnode --- struct zfetch --+--- struct zstream
                                 |
                                 +--- struct zstream
                                 |
                                 +--- struct zstream
```

Detecting interleaved reads

```
     index => current  stream
prev_index => previous stream
```

```
t0  ++
t1                                 ++++        [+] request page
t2    ++
t3                                     ++++
t4      ++
t5                                         ++++
t6        ++
t7                                             ++++
t8          ++
t9                                                 ++++
```

Detecting interleaved reads

```
     index => current  stream
prev_index => previous stream

if (probe_page(index - 1))
        /* handle interleaved reads */;
```

```
t0  ++
t1  --                              ++++        [+] request page
t2  --++                            ----        [-] cached  page
t3  ----                            ----++++
t4  ----++                          --------
t5  ------                          --------++++
t6  ------++                        ------------
t7  --------                        ------------++++
t8  --------++                      ----------------
t9  ----------                      ----------------++++
```

Thrashing safe readahead size

- `readahead_size = min(H - async_size, max)`
- H: cached referenced pages of the stream

history pages *H*

```
    l01 l11           l12   l21                    l22
   |-->|--->|      |------>|--->|               |------>|              LRU head
|  ====#====       ========#=====               ========#======               |
   |<---- L0 ------|<---------- L1 -------------|<----------- L2 -------------|
```

Let $f(l) = L$, where $l =$ visited pages, $L =$ enqueued pages in LRU in the mean while

$$\begin{aligned} f(l_{01}) &\leq L_0 \\ f(l_{11} + l_{12}) &= L_1 \\ f(l_{21} + l_{22}) &= L_2 \\ &\dots \\ f(l_{01} + l_{11} + \dots) &\leq Sum(L_0 + L_1 + \dots) \\ &\leq Length(\text{LRU}) = f(\text{thrashing threshold}) \end{aligned}$$

So the remained history pages $H = \sum l_{ij}$ is a conservative estimation of thrashing threshold.

Pagecache context based readahead

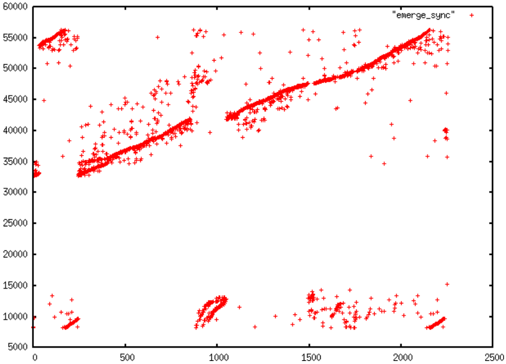
```
1  if (probe_page(index - 1)) {
2          begin = next_hole(index, max);
3          H     = index - prev_hole(index, 2*max);
4          end   = index + H;
5          update_window(begin, end);
6          submit_io();
7  }
```

```
    ==========================#____________..............
1                            ^index-1
2                             |----------->[begin
3  |<----------- H -----------|
4                             |----------- H ----------->]end
5                                          [ new window  ]

   [=] history [#] current [_] readahead [.] new readahead
```

Intermixed sequential+random reads

- each random read starts a new stream

algorithm	max streams	reasoning
Linux 2.6.22	1/fd	readahead is shutdown on any seek
Solaris/ZFS	32	the list of streams cannot grow large
context readahead	∞	radix tree lookups are near constant time



Matrix iterations

- seeky <u>column iterations</u> on huge matrix
  - transpose

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}^T$$

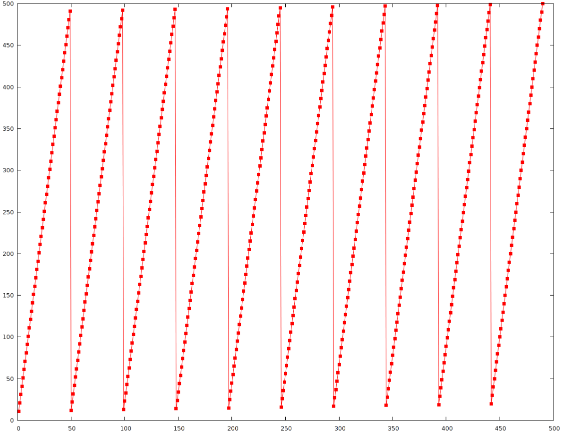
  - multiplication

$$\begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Matrix iterations: stride readahead

- regard as stride reads
- I/O made earlier but not larger
- sample I/O traces (*offset* + *size*):

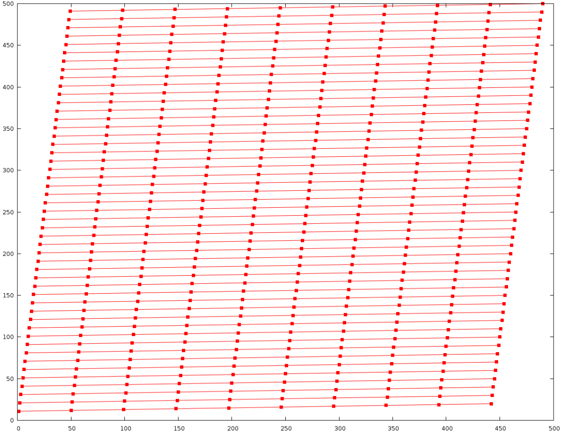
```
0+1,  1000+1,  2000+1,  3000+1,  ...,
1+1,  1001+1,  2001+1,  3001+1,  ...,
2+1,  1002+1,  2002+1,  3002+1,  ...
```



Matrix iterations: context readahead

- regard as concurrent streams
- large async I/O
- sample I/O traces (*offset* + *size*):

```
0+1,   1000+1,   2000+1,   3000+1,   ...,
1+4,   1001+4,   2001+4,   3001+4,   ...,
5+8,   1005+8,   2005+8,   3005+8,   ...
```



NFS server reads

1. client side doing big sequential read/readahead
2. send as small RPC requests
3. served at some random time by some random nfsd

Great trouble for readahead:

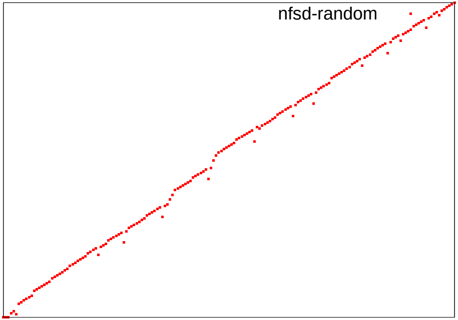
- read requests could be concurrent and out of order
- readahead state may not be available or up to date

8 nfsd serving different parts of a file

```
nfsd1 ++++                        ++++
nfsd2         ++++                            ++++
nfsd3                 ++++                                ++++
nfsd4             ++++                                ++++
nfsd5                         ++++        ++++
nfsd6     ++++                                    ++++
nfsd7                     ++++                                    ++++
nfsd8                                 ++++                    ++++
```

NFS server reads performance

- workload
  - NFS server, RAID, 100MB files
  - 16 nfsd doing <u>locally random</u>, <u>globally sequential</u> reads
- performance 1
  - vanilla: 30MB/s per disk
  - context: 60MB/s per disk
- performance 2
  - 2.6.18: 260MB/s
  - 2.6.23: 470MB/s (x1.8)



Sparse reads

- skip reading part of($\geq \frac{1}{8}$) a file
- `sequential = (index - prev_index <= 8 * req_size);`
- occur in
  - some divide-and-conquer algorithms
  - video editing on interleaved A/V files
- downside: may waste memory
- alternative: stride readahead

$\frac{1}{2}$ sparse reads

```
t0   ++++
t1           ++++
t2                   ++++
t3                           ++++
t4                                   ++++
t5                                           ++++
t6                                                  ++++
```

Sparse reads performance

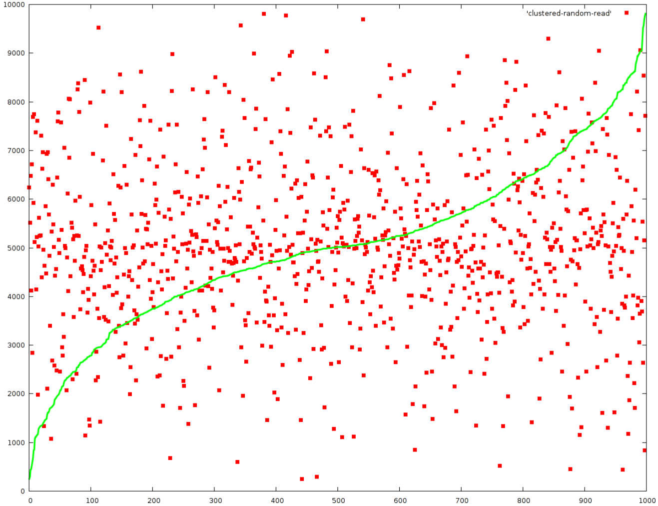
- workload
  - backup server: 24disks, hw-raid6, 11TB volume
  - `for(;;) { read(8KB); seek(8KB) }`
- performance
  - vanilla: 5MB/s
  - context: 200-250MB/s (x40-50)

parallel backups

```
t0   ++                              [+] process 1
t1       ++                          [+] process 2
t2           ++
t3     ++
t4               ++
t5         ++
t6                   ++
t7             ++
t8                       ++
t9                 ++
```

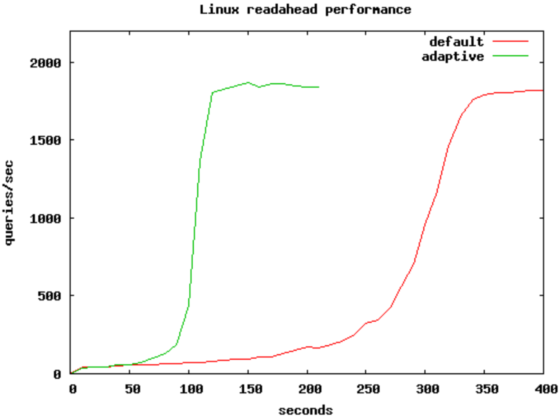
Clustered random reads

- where there is <u>hot area</u>, there is read-ahead/read-around
- context+sparse readahead algorithms are suitable



Clustered random reads performance

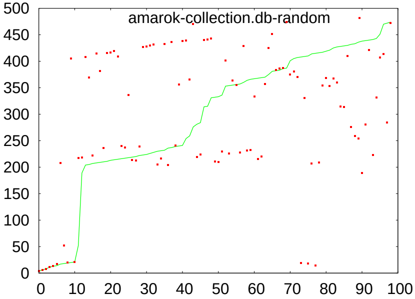
- randomly populating a big file into memory
  1. no noticeable regression on low density
  2. 3x speedup on high density

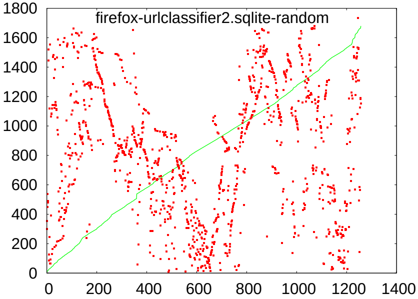


from: `http://www.linuxinsight.com/adaptive_readahead_benchmark.html`

Clustered random reads examples

- common in sqlite/postgresql/mysql workloads





Database performance

- sysbench OLTP with MySQL
  - up to 8% performance gain

- odbc-bench with Postgresql 7.4.11 on dual Opteron
  - vanilla: 92+99 tps
  - context: 113+125 tps (+25%)

- VACUUM ANALYZE on Postgresql
  - DB 1
    - vanilla: 11-12M ms
    - context: 1.7M ms (x7)
  - DB 2
    - vanilla: 300-500k ms
    - context: 150k ms (x2-3)

Thank you

This work is supported by Intel.