Reducing Seek Overhead with Application-Directed Prefetching

Steve VanDeBogart, Christopher Frost, Eddie Kohler
University of California, Los Angeles
http://libprefetch.cs.ucla.edu
### Disks are Relatively Slow

<table>
<thead>
<tr>
<th></th>
<th>Average Seek Time</th>
<th>Throughput</th>
<th>Whetstone Instr./Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1979</td>
<td>55 ms</td>
<td>0.5 MB/s</td>
<td>0.714 M</td>
</tr>
<tr>
<td>2009</td>
<td>8.5 ms</td>
<td>105 MB/s</td>
<td>2,057 M</td>
</tr>
<tr>
<td>Improvement</td>
<td>6.5 x</td>
<td>210 x</td>
<td>2,880 x</td>
</tr>
</tbody>
</table>

- **1979**: PDP 11/55 with an RL02 10MB disk
- **2009**: Core 2 with a Seagate 7200.11 500GB disk
Work Arounuds

- Buffer cache – Avoid redoing reads
- Write batching – Avoid redoing writes
- Disk scheduling – Reduce (expensive) seeks
- Readahead – Overlap disk & CPU time
Readahead

- Generally applies to sequential workloads
  - Harsh penalties for mispredicting accesses
  - Hard to predict nonsequential access patterns

- Some workloads are nonsequential
  - Databases
  - Image / Video processing
  - Scientific workloads: simulations, experimental data, etc.
Nonsequential Access

- Why so slow?
  - Seek costs

- Possible solutions
  - More RAM
  - More spindles
  - Disk scheduling

- Why are nonsequential access patterns often scheduled poorly?
  - Painful to get right
Example – Getting it Wrong

- Programmer will access nonsequential dataset

- Prefetch it
  
  `fadvise(fd, data_start, data_size, WILLNEED)`

- Now it's slower
  - Maybe prefetching evicted other useful data
  - Maybe the dataset is larger than the cache size
Libprefetch

- User space library
- Provides new prefetching interface
  - Application-directed prefetching
- Manages details of prefetching
- Up to 20x improvement
  - Real applications (GIMP, SQLite)
  - Small modifications (< 1,000 lines per app)
Libprefetch Contributions

• Microbenchmarks – Quantitatively understand problem

• Interface – Convenient interface to provide access information

• Kernel – Some changes needed

• Contention – Share resources
Outline

- Related work
- Microbenchmarks
- Libprefetch interface
- Results
Prefetching

- Determining future accesses
  - Historic access patterns
  - Static analysis
  - Speculative execution
  - Application-directed

- Using future accesses to influence I/O
Application-Directed Prefetching

- Patterson (Tip 1995), Cao (ACFS 1996)
- Roughly doubled performance
- Tight memory constraints
  - Little reordering of disk requests
- More in paper
Prefetching

Access pattern: 1, 6, 2, 8, 4, 7

No prefetching

CPU
I/O

Time
Prefetching

Access pattern: 1, 6, 2, 8, 4, 7

No prefetching

CPU I/O

Time

Traditional prefetching – Overlap I/O & CPU

CPU I/O
Prefetching

Access pattern: 1, 6, 2, 8, 4, 7

No prefetching

Traditional prefetching – Overlap I/O & CPU

Traditional prefetching – Fast CPU
Seek Performance

Average time to seek various distances
Seek Performance

Average time to seek various distances
Expensive Seeks

- Minimizing expensive seeks with disk scheduling – reordering

Access pattern: 1, 6, 2, 8, 4, 7

In order:

Reorder:
Reordering

- Must buffer out of order requests
- Reordering limited by buffer space
Reorder Prefetching

Access pattern: 1, 6, 2, 8, 4, 7

Traditional prefetching – Fast CPU

Reorder prefetching – Buffer size = 3

Reorder prefetching – Buffer size = 6
Random access to a 256MB file with varying amounts of reordering allowed
Buffer Size

Random access to a 256MB file with varying amounts of reordering allowed
Buffer Size

Graph showing the relationship between runtime and average seek distance with buffer size as a variable. The graph indicates that there is an optimal buffer size that minimizes runtime and seek distance for different storage capacities.
Random access to a 256MB file with varying amounts of reordering allowed
Buffer Size

- Buffer size important to performance
  - Too low: not using all capability, lower performance
  - Too high: evict useful data, performance goes down

- Start with all free and buffer cache memory
  - Libprefetch uses /proc to find free memory

- Change memory target with usage
More microbenchmarks

- Request size
  - Large requests vs. small requests

- Platter location
  - Start of disk vs. end of disk

- Infill
  - Reading extra data to eliminate small seeks
Libprefetch algorithm

- Application-directed prefetching for deep, accurate access lists
- Use as much memory as possible to maximize reordering
- Reorder requests to minimize large seeks
Interface Outline

- List of access entries
- Callback
  - Supply access list incrementally
  - Non-invasive to existing applications
Example

c = register_client(callback, NULL);

File A

0 450

File B

0 450
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);
Example

```
c = register_client();
r1 = register_region(c, A, 100, 0);
r2 = register_region(c, B, 300, 400);
r3 = register_region(c, B, 300, 400);
```

```
a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);
```

Access list entry:
file descriptor, file offset, marked flag

---

File A

```
0  75 350 450
```

File B

```
0  100 200 300 400 450
```
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

Flags: append, clear, complete

<table>
<thead>
<tr>
<th>File A</th>
<th>File B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  75  350  450</td>
<td>0  100  200  300  400  450</td>
</tr>
</tbody>
</table>
c = register_client(callback, NULL);

r1 = register_region(c, A, 75, 350);

r2 = register_region(c, B, 100, 200);

r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };

n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

Accepted entries "short" = full

File A

| 0 | 75 | 350 | 450 |

File B

| 0 | 100 | 200 | 300 | 400 | 450 |
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

fadvise(A, 100, WILL_NEED)
... fadvise(B, 150, WILL_NEED)
... fadvise(A, 200, WILL_NEED)

libprefetch_a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};
c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);

libprefetch_a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};
Example

```c
c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
```

File A

<table>
<thead>
<tr>
<th>0</th>
<th>75</th>
<th>350</th>
<th>450</th>
</tr>
</thead>
</table>

libprefetch_a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};

Check access list
Check in memory
`fincore(A, 100, ...)`
Example

c = register_client(callback, NULL);

r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
...
pread(B, ..., 350);

File A

| 0 | 75 | 350 | 450 |

File B

| 0 | 100 | 200 | 300 | 400 | 450 |

libprefetch_a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};

Access list doesn't match. Callback into application to update it.
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
...

void callback(void* arg, int markedFD, loff_t markedOffset,
              int requestedFD, loff_t requestedOffset);

libprefetch_a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
...

void callback(NULL, A, 100, B, 350) {
    a_list = compute_new-alist(B, 350);
    n = request_prefetching(c, a_list, 2, PF_SET | PF_DONE);
}

libprefetch_a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = {{A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1}};
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
...

void callback(NULL, A, 100, B, 350) {
    a_list = compute_new_a_list(B, 350);
    n = request_prefetching(c, a_list, 2, PF_SET | PF_DONE);
}

libprefetch_a_list = {{B, 150, 0}, ... {A, 200, 1}};
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ... {B, 150, 0}, ... {A, 200, 1} };

n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
...
pread(B, ..., 350);
...
pread(A, ..., 400);

libprefetch_a_list = {{B, 150, 0}, ... {A, 200, 1}};
Example

c = register_client(callback, NULL);
r1 = register_region(c, A, 75, 350);
r2 = register_region(c, B, 100, 200);
r3 = register_region(c, B, 300, 400);

a_list = { {A, 100, 1}, ..., {B, 150, 0}, ..., {A, 200, 1} };
n = request_prefetching(c, a_list, 3, PF_SET | PF_DONE);

pread(A, ..., 100);
...
pread(B, ..., 350);
...
pread(A, ..., 400);

libprefetch_a_list = {{B, 150, 0}, ..., {A, 200, 1}};

End of access list, callback to get more information.
Interface Summary

- **Access list**
  - Simply discloses application's intentions
  - Provided incrementally

- **Callback**
  - Asks application for more information
  - Easily retrofitted into existing applications
  - Aids in debugging access list information
Libprefetch

- Prefetching library

- A few important kernel modifications
  - fincore() - File page in memory?
  - Modified fadvise() - Fetch/evict file page

- Uses fadvise() to prefetch; manages details
  - When to prefetch
  - How much to prefetch
  - Right order for prefetching
Contention

- Disk scheduling – OS scheduler ok

- Memory for libprefetch behaves like bandwidth in TCP
  - Changes quickly
  - Performs poorly if over subscribed

- Use AIMD to determine memory target
  - Decrease when miss in buffer cache
  - Increase when all prefetched data stays in memory
Contention

- Disk scheduling – OS scheduler ok

- Memory for libprefetch behaves like bandwidth in TCP
  - Changes quickly
  - Performs poorly if over subscribed

- Use AIMD to determine memory target
  - Decrease when miss in buffer cache
  - Increase when all prefetched data stays in memory
Evaluation Methodology

- Pentium 4, 3.2GHz
- 512MB of RAM
- Seagate 7200.11 500GB SATA 3Gb/s
- Silicon Image 3132-2 SATA controller
- Logging over the network
Random Access

- SQLite with TPC-C like dataset:
  
  ```sql
  select * from Customer order by Zip_code;
  ```

- Secondary key => resulting rows will be randomly located in the dataset

- Total modifications: < 500 lines of code
Results: Random

- SQLite secondary key query

![Graph showing speedup vs. size of customer table plus zip code index (MB)]
Strided Accesses

- GIMP
  - Array of image tiles
  - Row-major layout accessed in Column-major order
  - Column-major layout accessed in Row-major order
  - Total modifications: 679 lines
Results: Strided

- GIMP blur
Sequential Access

- Sequentially read a large file
- Libprefetch should do just as well as readahead
Results: Sequential

![Graph showing speedup vs. amount of data read (MB)]
Impact of AIMD

The diagram compares the normalized runtime of Stock and Libprefetch under different levels of concurrency. The x-axis represents the number of concurrent GIMPs and concurrent SQLites, while the y-axis shows the normalized runtime with the single process set at 1.0.

- **Stock**
  - Concurrent GIMPs: 1.0, 2.12 s, 837 s
  - Concurrent SQLites: 67 s

- **Libprefetch**
  - Concurrent GIMPs: 1.0, 2.12 s, 756 s
  - Concurrent SQLites: 67 s
Performance with Contention

![Bar Chart]

- **Runtime (s)**
- **Stock GIMP**
- **Libprefetch GIMP**
- **Other (Stock)**

- **GIMP + md5sum**
  - Stock GIMP: 1300 seconds
  - Libprefetch GIMP: 700 seconds
  - Other (Stock): 600 seconds

- **GIMP + mem-walk**
  - Stock GIMP: 900 seconds
  - Libprefetch GIMP: 400 seconds
  - Other (Stock): 300 seconds
Conclusion

- A relatively simple library can transform accesses to avoid slow operations
- Microbenchmarks quantitatively show cause of nonsequential slowness
- Interface to easily retrofit applications
- Libprefetch handles kernel and concurrency complications
- Big performance gains (up to 20x) are possible for some workloads
Implementation Sketch

1. Scan access list – find enough entries to fill memory

2. `fadvise(DONT_NEED)` old entries

3. Sort new entries by file offset

4. `fadvise(WILL_NEED)` new entries

5. Return to intercepted read