Anticipatory Scheduling
A disk scheduling framework to overcome deceptive idleness in synchronous I/O

Kim Sang Eon
July 26, 2010
SUNG KYUN KWAN Univ.
SAMSUNG ELECTRONICS CO., LTD.
Anticipatory Scheduling
A disk scheduling framework to overcome deceptive idleness in synchronous I/O

anticipatory
예상하고서, 예상하고 앞질러 하는
선행하는

Deceptive idleness
기만하는, 현혹하는 게으름

Synchronous I/O
동기식 I/O  ex) read
⇔ asynchronous  ex) write
Why I/O scheduler

- Disk seek is the slowest operation in a computer
  - A system would perform horribly without a suitable I/O scheduler

- Improve overall disk throughput by
  - Reorder request to reduce the disk seek time
  - Merge request to reduce the number of requests

- Prevent starvation
  - Submit request before deadline
  - Avoid read starvation by write

- Provide fairness among different processes
Anticipatory scheduling Background

- Disk Schedulers reorder available disk request for:
  - performance by seek optimization
  - proportional resource allocation

- Any policy needs multiple outstanding request to make good decisions!
With enough request

issued by process A

issued by process B

seek

location on disk

time
With Synchronous I/O

- **Deceptive Idleness**
  - Process A is about to issue next request
  - But scheduler hastily assumes that process A has no further request
Proportional scheduler

Allocate disk service in say 1:2 ratio:

Deceptive idleness causes 1:1 allocation:
Proportional scheduler
Prefetching

- Asynchronous prefetch
  - Partially work around
  - Predicting the future request issue pattern
  - Limitation

- Application-driven prefetch
  - Incapable of issuing accurate prefetch request
  - Have to be rewritten
  - More data copying and cache pollution
  - Not enabled in some operating systems

- Kernel-driven prefetch
  - Less capable of predicting future access patterns
  - Penalties of misprediction can be high
Anticipatory Scheduling

- **Key idea**
  - Briefly wait for process whose request was last serviced

- **Work-conserving**
  - If the disk is idle or a request is completed,
  - Next request in the queue is scheduled immediately

- **Non-work-conserving**
  - The disk stands idle in the face of nonempty queue
How long to wait

- Benefit of waiting > Cost of keeping idle
- Need Heuristics !!
Anticipatory Scheduling Framework

Various Kernel Subsystems

Disk requests

Anticipatory scheduling core

Scheduling policy implementation

Anticipation heuristic

Disk Driver
Benefit-Cost Analysis

- Seek reducing scheduler

```plaintext
benefit = (calculate_positioning_time(Candidate) - LP.expected_positioning_time)
cost = max(0, LP.expected_median_thinktime - elapsed)
waiting_duration = max(0, LP.expected_95percentile_thinktime - elapsed)
return (benefit > cost ? waiting_duration : 0)
```
Benefit-Cost Analysis

- Proportional scheduler

- Conditions of Waiting for process whose req was last serviced
  - If it has received less than its allocation
  - If it has thinktime below a threshold (ex. 3ms)
Evaluation

- Microbenchmark: Access patterns
Evaluation

- Microbenchmark: Varying thinktimes

![Diagram showing throughput vs. thinktime for symmetric and asymmetric processes.](image-url)
Proportional-share Scheduling
Cooperative Anticipatory Scheduler

- Proposed in this paper
  - Enhancements to Linux I/O scheduler, Seetharami Seelam, OLS2005

- The problems of anticipatory Assumption
  - Synchronous disk req are issued by individual processes
  - For anticipation to work properly, the anticipated process must be alive

- The problems of anticipatory scheduler
  - Anticipation works only when req are issued by same process
The problems of anticipatory scheduler

- Concurrent streaming and chunk read programs.
  - By assume 1, problem occure
  - Program A: synchronous read request by a single process
  - Program B: dependent chunk read by a different process
    ex) shell script
  - Anticipation works only on a per-process basis (Program A)

- Concurrent execution of these two programs results in starvation of processes generated by Program B.
Concurrent chunk read programs.

- By assume 2, problem occur
- Program B: dependent chunk read by a different process
- Process1 and Process2 will work well.
- Once a process1 terminates (with pending request)
- Dispatch a request of the Process2

- If there are F files, at least 2F-1 seeks are necessary
- Seek storms. Poor throughput
CAS solution

Solution
- Keep anticipating even when the anticipated process has exited
- Cooperative exit probability
- Existence of cooperative processes related to dead processes
CAS Evaluation

Streamlining and chunk reads

- Anticipation works well for A
- But, what happens with B
- B: inadmissible time using AS
- CAS provides anticipation on a per-group basis; thus seeks reduced and throughput improved

Program A:
```
while true
do
   cat big-file > /dev/null
done
```

Program B:
```
time find . -type f -exec \cat '{}' ';' > /dev/null
```

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Execution Time (sec.)</th>
<th>Throughput (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>297</td>
<td>9</td>
</tr>
<tr>
<td>LAS</td>
<td>4767</td>
<td>35</td>
</tr>
<tr>
<td>CAS</td>
<td>255</td>
<td>34</td>
</tr>
</tbody>
</table>
Multiple chunk reads

- Two instances of chunk read to disjoint disk blocks
- Anticipation fail to both
- Results for reading Linux source tree
- There is some seeking, but CAS does not seek as much as others
Deadline I/O scheduler

- **Goal**
  - Reorder request to improve I/O performance while simultaneously ensuring that no I/O request is being starved

- **Each request is associated with a expire time.**
  - Read: 500ms, Write: 5sec

- **Request are inserted into**
  - A sorted-by-start-sector queue (two queues, for read and write)
  - A FIFO list (two list) sorted by expire time

- **Normally, request are pulled from sorted queues. However, if the request at the head of either FIFO queue expires, requests are still processed in sorted order but started from the first request in the FIFO queue**
Architecture of Deadline I/O scheduler

- The sorted queues are built on red-black trees.
- For back merge purpose, requests are hashed into an array of list indexed by the end sector.

The diagram shows the process flow of the deadline I/O scheduler:

1. Request hash table (sorted by end sector)
2. Read RB tree (sorted by start sector)
3. Read FIFO lists (sorted by queue time)
4. Write RB tree (sorted by start sector)
5. Write FIFO lists (sorted by queue time)
6. Dispatch queue
7. Device driver

The functions involved are:
- deadline_insert_request
- deadline_dispatch_requests
- deadline_next_request
Goal

- Provide fair allocation of I/O bandwidth among all the initiators of I/O request.

CFQ can be configured to provide fairness at per process, per process group, per user and per user group levels.

Each initiator has its own request queue and CFQ services these queues round-robin.
Architecture of CFQ

- `cfq_insert_request`
- `queue hash by tgid`
- `tgid 1 queue`
- `tgid 2 queue`
- `tgid n queue`
- `Round robin serving 1 request at a time`
- `cfq_dispatch_requests`
- `device queue (sorted by sector)`

- Red-black tree (sorted by sector)
- Read FIFO lists (sorted by queue time)
- Write FIFO lists (sorted by queue time)
Evaluation of each scheduler

- AS has execution time comparable to CAS in Web server
- CAS has best in Mail server
- CAS has best in File server
- CFQ has best in Meta data
Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O, SitaramIyer, ACM SOSP’01

Enhancements to Linux I/O scheduling, SeetharamiSeelam, OLS’05

Research.utep.edu/Portals/937/SeelamLinuxSymposium2005.pdf