FAST: Quick Application Launch on Solid-State Drives

Yongsoo Joo¹, Junhee Ryu², Sangsoo Park¹, and Kang G. Shin¹,³

¹Ewha Womans University, Korea
²Seoul National University, Korea
³University of Michigan, USA
Application Launch Delay

● Elapsed time between two events
  ● A user clicks the icon
  ● The application becomes responsible

● Important for interactive applications
  ● Critically affects user satisfaction
Moore’s law not applicable
- Faster CPU and larger main memory not helpful
- HDD seek and rotational latencies do not improve well

Exponential improvement
Linear improvement
Application Launch Performance

- Application launch breakdown

![Bar chart showing computation, seek rotational latency, and data transfer time for F-Spot, Firefox, Evolution, Totem, and Rhythmbox.](image-url)
SW-Level Optimization

- Many SW-level schemes deployed in OSes
  - Application defragment, Superfetch, readahead, BootCache, etc.

- Sorted prefetch (ex: Windows prefetch)
  - Obtain the set of accessed blocks for each application
    - Monitor I/O requests during an application launch
  - Pause the target application upon detection of its launch
  - Prefetch the predetermined set of blocks in their LBA order
    - Reduce the total seek distance of the disk head
  - Resume the launch after the prefetch completes
SW-Level Optimization

- How sorted prefetch works
Flash-based SSD

- The single most effective way to eliminate disk head positioning delay
  - Acrobat reader: 4.0s -> 0.8s (84% reduction)
  - Matlab: 16.0s -> 5.1s (68% reduction)

- Characteristics
  - Consist of multiple NAND flash chips
  - No mechanical moving part
  - Uniform access latency (a few 100 microseconds)

- Prices now affordable
  - 80 GB MLC SSD: less than 200$ now
Motivation

**Question:** Are we satisfied with the app launch on SSD?

- **Yes** for lightweight applications (e.g., less than 1 sec)
- **No** for heavy applications (e.g., more than 5 sec)
  - Far from ultimate user satisfaction
  - Faster application launch is always good (at least, not bad)

**Needs increase for launch optimization on SSDs**

- Applications are getting **HEAVIER**
  - More blocks to be read
- SSD random read performance improves slowly
  - Bounded by the single chip performance
HDD-Aware Optimizers on SSD

**Question:** Will traditional HDD optimizers work for SSDs?
- Consensus: they will not be effective on SSDs
- Rationale: they mostly optimize disk head movement
  - No disk head in SSDs
  - Often recommended not to use on SSDs

**Microsoft Windows 7**
- HDD-aware optimizers disabled upon detection of SSD
  - Windows prefetch, Application defragmentation, Superfetch, Readyboost, etc.
Sorted Prefetch on SSDs

- No benefit from LBA sorting
  - Uniform seek latency of SSD
- Launch performance still improves
  - Increased effective queue depth (0.3->3.4, app: Eclipse)
  - Observed 7% launch time reduction: better than nothing!

![Graphs and plots](image-url)
FAST: Fast Application STarter

- Overlap CPU computation with SSD accesses

(a) Cold start scenario

(b) Warm start scenario

(c) Proposed prefetching ($t_{cpu} > t_{ssd}$)
Application Launch Sequence

- **Deterministic** block requests over repeated launches
- Raw block request traces

- Application launch sequence

  Block requests irrelevant to the application launch
What to Do

• Application launch sequence profiling
  • Using `blktrace` tool

• Prefetcher generation
  • Replay block requests according to the application launch sequence

• Prefetcher execution
  • Simultaneously with the original application
  • By wrapping the system call `exec()`
    • `LD_PRELOAD`
Prefetcher Generation

- Example application launch sequence
  - AB->C->D

- Block-level I/O: (start LBA, size)
  - (5, 2)->(1, 1)->(7, 1) <- obtainable from blktrace

```
<table>
<thead>
<tr>
<th>LBA</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"/dev/sda"

Accessed block
```
Prefetcher Generation

- Example application launch sequence
  - AB->C->D

- Block-level I/O: (start LBA, size)
  - (5, 2)->(1, 1)->(7, 1) <- obtainable from blktrace

- File-level I/O: (filename, offset, size)
  - ("b.so", 2, 2)->("a.conf", 1, 1)->("c.lib", 0, 1)

```
<table>
<thead>
<tr>
<th>File offset</th>
<th>&quot;a.conf&quot;</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBA</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Accessed block</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File offset</th>
<th>&quot;b.so&quot;</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBA</td>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Accessed block</td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File offset</th>
<th>&quot;c.lib&quot;</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBA</td>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Accessed block</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

"/dev/sda"
Prefetcher Generation

- Block-level I/O replay

```c
int main(void) {
    fd = open("/dev/sda",O_RDONLY|O_LARGEFILE);
    posix_fadvise(fd, 5*512, 2*512, POSIX_FADV_WILLNEED);
    posix_fadvise(fd, 1*512, 1*512, POSIX_FADV_WILLNEED);
    posix_fadvise(fd, 7*512, 1*512, POSIX_FADV_WILLNEED);
    return 0;
}
```

![Diagram showing LBA and size of files and their accessed blocks]
Page Cache Structure

Page cache

inode
/dev/sda

cached blocks
A B
C D

a.conf b.so c.lib
Page Cache Structure

Page cache

inode /dev/sda

cached blocks

A B

C D

a.conf Miss!
b.so Miss!
c.lib Miss!
Page Cache Structure

What we need to construct
Prefetcher Generation

- File-level I/O replay

```c
int main(void) {
    fd1 = open("b.so", O_RDONLY);
    posix_fadvise(fd1, 2*512, 2*512, POSIX_FADV_WILLNEED);
    fd2 = open("a.conf", O_RDONLY);
    posix_fadvise(fd2, 1*512, 1*512, POSIX_FADV_WILLNEED);
    fd3 = open("c.lib", O_RDONLY);
    posix_fadvise(fd3, 0*512, 1*512, POSIX_FADV_WILLNEED);
    return 0;
}
```

<table>
<thead>
<tr>
<th>file name</th>
<th>file offset</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a.conf&quot;</td>
<td>0 (C)</td>
<td>1</td>
</tr>
<tr>
<td>&quot;b.so&quot;</td>
<td>1 (A)</td>
<td>2</td>
</tr>
<tr>
<td>&quot;c.lib&quot;</td>
<td>2 (B)</td>
<td>3</td>
</tr>
</tbody>
</table>

LBA:
```
0 1 2
3 4 5 6 7 8 9
```

"/dev/sda"

- Accessed block:

Red rectangles indicate accessed blocks.
Block-to-File Level I/O Conversion

- LBA-to-inode mapping
  - Not supported by EXT file system

(5,2)  (“b.so”, 2,2)
(1,1)  (“a.conf”, 1,1)
(7,1)  (“c.lib”, 0,1)

```
File offset
"a.conf"
0 1 2
C
"b.so"
0 1 2 3
A B
"c.lib"
0 1 2
D
LBA
0 1 2
3 4 5 6
7 8 9
"/dev/sda"
```

Accessed block
Block-to-File Level I/O Conversion

- Inode-to-LBA map for a single file
  - Easy to build

- LBA-to-inode map for the entire file system
  - Millions of files in a file system
  - Frequently changed
  - Only a few 100s of files used by a single application

- Our approach: build a partial map for each application
  - Determine the set of files used for the launch
    - Monitoring system calls using filename as their argument
Application Prefetcher

- Automatically generated application prefetcher for Gimp

```c
int main(void) {
    ... 
    readlink("/etc/fonts/conf.d/90-ttf-arphic-uming-embolden.conf", linkbuf, 256);
    int fd423;
    fd423 = open("/etc/fonts/conf.d/90-ttf-arphic-uming-embolden.conf", O_RDONLY);
    posix_fadvise(fd423, 0, 4096, POSIX_FADV_WILLNEED);
    posix_fadvise(fd351, 286720, 114688, POSIX_FADV_WILLNEED);
    int fd424;
    fd424 = open("/usr/share/fontconfig/conf.avail/90-ttf-arphic-uming-embolden.conf", O_RDONLY);
    posix_fadvise(fd424, 0, 4096, POSIX_FADV_WILLNEED);
    int fd425;
    fd425 = open("/root/.gnupg/trustdb.gpg", O_RDONLY);
    posix_fadvise(fd425, 0, 4096, POSIX_FADV_WILLNEED);
    dirp = opendir("/var/cache/");
    if(dirp)while(readdir(dirp));
    ... 
    return 0;
}
```
CPU and SSD Usage

- Cold start
  - CPU
  - SSD
- Warm start
  - CPU
  - SSD
  (a)
  (b)
- FAST
  - CPU
  - SSD
- Sorted prefetch
  - CPU
  - SSD

Low CPU usage

Reduction: 24%
CPU and SSD Usage

Cold start
CPU SSD
Warm start
CPU SSD
(a) (b)
FAST
CPU SSD
Sorted prefetch
CPU SSD
Low CPU usage
Eclipse
0 1 2 3 4 5
Reduction: 24%
Firefox
0 1
Reduction: 37%
CPU and SSD Usage

Cold start
- CPU
- SSD

Warm start
- CPU
- SSD
- (a)
- (b)

FAST
- CPU
- SSD

Sorted prefetch
- CPU
- SSD

Low CPU usage

Eclipse

Firefox

Reduction: 24%

Reduction: 37%
**Measured Application Launch Time**

- **Launch time reduction**
  - Warm start: 37% (upper bound)
  - Proposed: 28% (min: 16%, max: 46%)
  - Sorted prefetch: 7% (min: -5%, max: 21%)

(Normalized to the cold start time.)
Measured Application Launch Time

- Launch time reduction
  - Warm start: 37% (upper bound)
  - Proposed: 28% (min: 16%, max: 46%)
  - Sorted prefetch: 7% (min: -5%, max: 21%)
Applicability on Smartphones

• Similarity to PCs with a SSD
  • Running various applications
    • Application launch performance does matter
  • NAND Flash-based storage
    • The same performance characteristic as SSDs
  • Slightly modified OSes and file systems designed for PCs
    • Easy to port
Applicability on Smartphones

• Further benefits
  • More frequent launches of applications
  • Limited main memory capacity
    • Cold start scenario occurs more often
  • Slower CPU and flash storage speed
    • Relatively longer application launch time
Applicability on Smartphones

- Measured cold & warm start time on iPhone 4
  - Average cold start time: 6.1 seconds
  - Warm start time: 63% of cold start time
Introduction an application prefetcher designed for SSDs

Our ultimate goal

- **Warm start** performance in the **cold start** scenario

- Further improving FAST by exploiting the SSD parallelism

- See our poster!
Backup Slides
Applicability on HDDs

- FAST works as well on HDDs, but ...
  - Application launch on HDDs: I/O bound
  - Little room for overlapping CPU time and HDD access time
  - Launch time reduction: 15%

- Sorted prefetch performs better
  - Launch time reduction: 40%

![Bar chart showing normalized application launch time on HDD. Cold: 100%, FAST: 85%, Sorted: 60%, Warm: 15%.]
Determinism on Multi-Core

- We observed determinism even on multi-core CPUs
  - Only one core is active during the most time periods
  - SSD is mostly idle when two or more cores are active
Why not Capturing File I/O?

- Why not simply capture all the file-level I/Os and replay them?
  - Ex) Capture all read() calls using strace
- That’s possible, but the problem is...
  - The number of read() calls are extremely large
  - Only few of them will cause page fault, generating a block I/O
  - Replaying all the captured read() calls are inefficient
    - In terms of both prefetcher size and function call overhead
  - Not easy to determine which of them will actually cause page faults
    - May be more complicated than our approach (block-level to file-level I/O conversion)