Joo Y, Ryu J, Park S *et al.* Improving application launch performance on solid state drives. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(4): 727–743 July 2012. DOI 10.1007/s11390-012-1259-3

Improving Application Launch Performance on Solid State Drives

Yongsoo Joo[1], *Member*, *IEEE*, Junhee Ryu[2], Sangsoo Park[1,*], *Member*, *IEEE*, and Kang G. Shin[1,3], *Fellow*, *IEEE*

[1] *Department of Computer Science and Engineering, Ewha Womans University, 11-1 Daehyun-dong Seodaemun-gu Seoul 120-750, Korea*

[2] *Department of Computer Science and Engineering, Seoul National University, 599 Kwanak-Gu Kwanak Rd. Seoul 151-744, Korea*

[3] *Department of Electrical Engineering and Computer Science, University of Michigan, 2260 Hayward St., Ann Arbor MI 48109, U.S.A.*

E-mail: ysjoo@ewha.ac.kr; jhryu@cslab.snu.ac.kr; sangsoo.park@ewha.ac.kr; kgshin@eecs.umich.edu

Received November 18, 2011; revised May 10, 2012.

Abstract Application launch performance is of great importance to system platform developers and vendors as it greatly affects the degree of users' satisfaction. The single most effective way to improve application launch performance is to replace a hard disk drive (HDD) with a solid state drive (SSD), which has recently become affordable and popular. A natural question is then whether or not to replace the traditional HDD-aware application launchers with a new SSD-aware optimizer. We address this question by analyzing the inefficiency of the HDD-aware application launchers on SSDs and then proposing a new SSD-aware application prefetching scheme, called the Fast Application STarter (FAST). The key idea of FAST is to overlap the computation (CPU) time with the SSD access (I/O) time during an application launch. FAST is composed of a set of user-level components and system debugging tools provided by Linux OS (operating system). Hence, FAST can be easily deployed in any recent Linux versions without kernel recompilation. We implement FAST on a desktop PC with an SSD running Linux 2.6.32 OS and evaluate it by launching a set of widely-used applications, demonstrating an average of 28% reduction of application launch time as compared to PC without a prefetcher.

Keywords application launch performance, I/O prefetch, solid state drive

1 Introduction

Application launch performance is one of the important metrics for the design or selection of a desktop or a laptop PC as it critically affects the user-perceived performance. Unfortunately, application launch performance has not kept up with the remarkable progress of CPU performance that has thus far evolved according to Moore's law. As frequently-used or popular applications get "heavier" (by adding new functions) with each new release, their launch takes longer even if a new, powerful machine equipped with high-speed multi-core CPUs and several GBs of main memory is used. This undesirable trend is known to stem from the poor random access performance of hard disk drives (HDDs). When an application stored in an HDD is launched, up to thousands of block requests are sent to the HDD, and a significant portion of its launch time is spent on moving the disk head to proper track and sector positions, i.e., *seek* and *rotational* latencies. Unfortunately, the HDD seek and rotational latencies have not been improved much over the last few decades, especially compared to the CPU speed improvement. In spite of the various optimizations proposed to improve the HDD performance in launching applications, users must often wait tens of seconds for the completion of launching frequently-used applications, such as Windows Outlook.

A quick and easy solution to eliminate the HDD's seek and rotational latencies during an application launch is to replace the HDD with a solid state drive (SSD). An SSD consists of a number of NAND flash chips, and does not use any mechanical parts, unlike disk heads and arms of a conventional HDD. While the HDD access latency — which is the sum of seek and rotational latencies — ranges up to a few tens of milliseconds (ms), depending on the seek distance, the SSD shows a rather uniform access latency of about a few

Regular Paper

This research was supported by RP-Grant 2010 of Ewha Womans University.

An earlier version of this paper was presented at the USENIX Conference on File and Storage Technologies (FAST) 2011 under the title "FAST: Quick application launch on solid-state drives."

*Corresponding Author

hundred micro-seconds (*us*). Replacing an HDD with an SSD is, therefore, the single most effective way to improve application launch performance.

Until recently, using SSDs as the secondary storage of desktops or laptops has not been an option for most users due to the high cost-per-bit of NAND flash memories. However, the rapid advance of semiconductor technology has continuously driven the SSD price down, and at the end of 2010, the price of an 80 GB SSD has fallen below 200 US dollars. Furthermore, SSDs can be installed in existing systems without additional hardware or software support because they are usually equipped with the same interface as HDDs, and operating systems (OSes) see an SSD as a block device just like an HDD. Thus, end-users begin to use an SSD as their system disk to install the OS image and applications.

Although an SSD can significantly reduce the application launch time, it does not give users ultimate satisfaction for all applications. For example, using an SSD reduces the launch time of a heavy application from tens of seconds to several seconds. However, users will soon become used to the SSD launch performance, and will then want the launch time to be reduced further, just as they see from light applications. Furthermore, users will keep on adding functions to applications, making them heavier with each release and their launch time greater. According to a recent report[1], the growth of software is rapid and limited only by the ability of hardware. These call for the need to further improve application launch performance on SSDs.

Unfortunately, most previous application launch optimizers are intended for HDDs and have not accounted for the SSD characteristics. Furthermore, some of them may rather be detrimental to SSDs. For example, running a disk defragmentation tool on an SSD is not beneficial because changing the physical location of data in the SSD does not affect its access latency. Rather, it generates unnecessary write and erase operations, thus shortening the SSD's lifetime.

In view of these, the first step toward SSD-aware optimization may be to simply disable the traditional optimizers designed for HDDs. For example, Windows 7 disables many functions, such as disk defragmentation, application prefetch, Superfetch, and Readyboost when it detects an SSD being used as a system disk[2]. Let consider another example. Linux is equipped with four disk I/O schedulers: NOOP, anticipatory, deadline, and completely fair queueing. The NOOP scheduler almost does nothing to improve HDD access performance, thus providing the worst performance on an HDD. Surprisingly, it has been reported that NOOP shows better performance than the other three sophisticated schedulers on an SSD[3].

To the best of our knowledge, this is the first attempt to focus entirely on improving application launch performance on SSDs. Specifically, we propose a new application prefetching method, called the Fast Application STarter (FAST), to improve application launch time on SSDs. The key idea of FAST is to overlap the computation (CPU) time with the SSD access (I/O) time during each application launch. To achieve this, we monitor the sequence of block requests in each application, and launch the application simultaneously with a prefetcher that generates I/O requests according to the *a priori* monitored application's I/O request sequence. FAST consists of a set of user-level components, a system-call wrapper, and system debugging tools provided by Linux OS. FAST can be easily deployed in most recent Linux versions without kernel recompilation. We have implemented and evaluated FAST on a desktop PC with an SSD running Linux 2.6.32, demonstrating an average of 28% reduction of application launch time as compared to PC without a prefetcher.

This paper makes the following contributions:

- qualitative and quantitative evaluation of the inefficiency of traditional HDD-aware application launch optimizers on SSDs;
- development of a new SSD-aware application prefetching scheme, called FAST; and
- implementation and evaluation of FAST, demonstrating its superiority and deployability.

The paper is organized as follows. In Section 2, we review other related efforts and discuss their performance in optimizing application launch on SSDs. Section 3 describes the key idea of FAST and presents a lower bound of the application launch time achievable with FAST. Section 4 discusses design issues to implement FAST. Section 5 details the implementation of FAST on Linux OS, while Section 6 evaluates its performance using various real-world applications. Section 7 discusses the applicability of FAST and Section 8 compares FAST with traditional I/O prefetching techniques. We conclude the paper with Section 9.

2 Background

In this section, we summarize traditional application launch optimization techniques for HDDs as well as recent work for SSD performance optimization. We also discuss the effectiveness of the traditional HDD-based launch optimization schemes on SSDs.

2.1 Application Launch Optimization

Application developers are usually advised to optimize their applications for fast startup. For example,

they may be advised to postpone loading non-critical functions or libraries so as to make applications respond as fast as possible[4-5]. They are also advised to reduce the number of symbol relocations while loading libraries, and to use dynamic library loading. There have been numerous case studies — based on in-depth analyses and manual optimizations — of various target applications/platforms, such as Linux desktop suite platform[6], a digital TV[7], and a digital still camera[8]. However, such an approach requires the experts' manual optimizations for each and every application. Hence, it is economically infeasible for general-purpose systems with many (dynamic) application programs.

A snapshot boot technique has also been suggested for fast startup of embedded systems[9], which is different from the traditional hibernate shutdown function in that a snapshot of the main memory after booting an OS is captured only once, and used repeatedly for every subsequent booting of the system. However, applying this approach for application launch is not practical for the following reasons. First, the page cache in main memory is shared by all applications, and separating only the portion of the cache content that is related to a certain application is not possible without extensive modification of the page cache. Furthermore, once an application is updated, its snapshot should be invalidated immediately, which incurs runtime overhead.

Modern desktops are equipped with large (up to several GBs) main memory, and often have abundant free space available in the main memory. Prediction-based prefetching, such as Superfetch[10] and Preload[11], loads an application's code blocks in the free space even if the user does not explicitly express his/her intent to execute that particular application. These techniques monitor and analyze the users' access patterns to predict which applications to be launched in future. Consequently, the improvement of launch performance depends strongly on prediction accuracy.

Windows OS is equipped with an application prefetcher[12] that prefetches application code blocks in a sorted order of their logical block addresses (LBAs) to minimize disk head movements. A similar idea has also been implemented for Linux OS[13-14]. We call these approaches *sorted prefetch*. It monitors HDD activities to maintain a list of blocks accessed during the launch of each application. Upon detection of an application launch, the application prefetcher immediately pauses its execution and begins to fetch the blocks in the list in an order sorted by their LBAs. The application launch is resumed after fetching all the blocks, and hence, no page miss occurs during the launch.

The block list information can also be used in a different way to further reduce the seek distance during an application launch. Modern OSes commonly support an HDD defragmentation tool that reorganizes the HDD layout so as to place each file in a contiguous disk space. In contrast, the defragmentation tool can relocate the blocks in the list of each application by their access order[12], which helps reduce the total HDD seek distance during the launch.

2.2 SSD Performance Optimization

SSDs have become affordable and begun to be deployed in desktop and laptop PCs, but their performance characteristics have not yet been understood well. So, researchers conducted in-depth analyses of their performance characteristics, and suggested ways to improve their runtime performance. Extensive experiments have been carried out to understand the performance dynamics of commercially-available SSDs under various workloads, without knowledge of their internal implementations[15]. Also, SSD design space has been explored and some guidelines to improve the SSD performance have been suggested[16]. A new write buffer management scheme has also been suggested to improve the random write performance of SSDs[17]. Traditional I/O schedulers optimized for HDDs have been revisited in order to evaluate their performance on SSDs, and then a new I/O scheduler optimized for SSDs has been proposed[3,18].

2.3 Launch Optimization on SSDs

As discussed in Subsection 2.1, various approaches have been developed and deployed to improve the application launch performance on HDDs. On one hand, many of them are effective on SSDs as well, and orthogonal to FAST. For example, application-level optimization and prediction-based prefetch can be used together with FAST to further improve application launch performance.

On the other hand, some of them exploit the HDD characteristics to reduce the seek and rotational delay during an application launch, such as the sorted prefetch and the application defragmentation. Such methods are ineffective for SSDs because the internal structure of an SSD is very different from that of an HDD. An SSD typically consists of multiple NAND flash memory modules, and does not have any mechanical moving part. Hence, unlike an HDD, the access latency of an SSD is irrelevant to the LBA distance between the last and the current block requests. Thus, prefetching the application code blocks according to the sorted order of their LBAs or changing their physical locations will not make any significant performance

improvement on SSDs. As the sorted prefetch has the most similar structure to FAST, we will quantitatively compare its performance with FAST in Section 6.

3 Application Prefetching on SSDs

This section illustrates the main idea of FAST with examples and derives a lower bound of the application launch time achievable with FAST.

3.1 Cold and Warm Starts

We focus on the performance improvement in case of a cold start, or the first launch of an application upon system bootup, representing the worst-case application launch performance. Fig.1(a) shows an example cold start scenario, where s_i is the i-th block request generated during the launch and n the total number of block requests. After s_i is completed, the CPU proceeds with the launch process until another page miss takes place. Let c_i denote this computation.

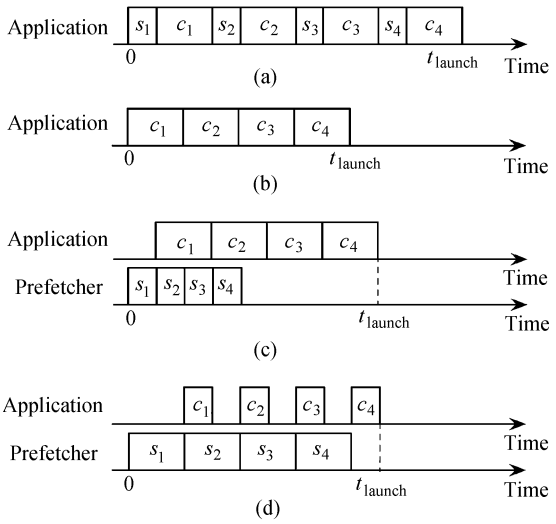


Fig.1. Various application launch scenarios ($n = 4$). (a) Cold start scenario. (b) Warm start scenario. (c) Proposed prefetching ($t_{CPU} > t_{SSD}$). (d) Proposed prefetching ($t_{CPU} < t_{SSD}$).

The opposite extreme is a warm start where all the code blocks necessary for launch have been found in the page cache, and thus, no block request is generated, as shown in Fig.1(b). This occurs when the application launches again shortly after its closure. The warm start represents a lower-bound of the application launch time achievable with optimization of the secondary storage.

Let the time spent for s_i and c_i be denoted by $t(s_i)$ and $t(c_i)$, respectively. Then, the computation (CPU) time, t_{CPU}, is expressed as

$$t_{CPU} = \sum_{i=1}^{n} t(c_i), \tag{1}$$

and the SSD access (I/O) time, t_{SSD}, is expressed as

$$t_{SSD} = \sum_{i=1}^{n} t(s_i). \tag{2}$$

3.2 Proposed Application Prefetcher

The rationale behind FAST is that the I/O request sequence generated during an application launch does not change over repeated launches of the application in case of cold-start. The determinism in random read requests has been also observed and exploited by many previous work[19-21].

The key idea of FAST is to overlap the SSD access (I/O) time with the computation (CPU) time by running the application prefetcher concurrently with the application itself. The application prefetcher replays the I/O request sequence of the original application, which we call an application launch sequence. An application launch sequence S can be expressed as $(s_1, \ldots, s_n)$.

Fig.1(c) illustrates how FAST works, where $t_{CPU} > t_{SSD}$ is assumed. At the beginning, the target application and the prefetcher start simultaneously, and compete with each other to send their first block request to the SSD. However, the SSD always receives the same block request s_1 regardless of which process gets the bus grant first. After s_1 is fetched, the application can proceed with its launch by the time $t(c_1)$, while the prefetcher keeps issuing the subsequent block requests to the SSD. After completing c_1, the application accesses the code block corresponding to s_2, but no page miss occurs for s_2 because it has already been fetched by the prefetcher. It is the same for the remaining block requests, and thus, the resulting application launch time t_{launch} becomes

$$t_{launch} = t(s_1) + t_{CPU}. \tag{3}$$

Fig.1(d) shows another possible scenario where $t_{CPU} < t_{SSD}$. In this case, the prefetcher cannot complete fetching s_2 before the application finishes computation c_1. However, s_2 can be fetched by $t(c_1)$ earlier than that of the cold start, and this improvement is accumulated for all of the remaining block requests, resulting in t_{launch}:

$$t_{launch} = t_{SSD} + t(c_n). \tag{4}$$

Note that n ranges up to a few thousands for typical applications, and thus, $t(s_1) \ll t_{CPU}$ and $t(c_n) \ll t_{SSD}$. Consequently, (3) and (4) can be combined into a single equation as:

$$t_{launch} \approx \max(t_{SSD}, t_{CPU}), \tag{5}$$

which represents a lower bound of the application launch time achievable with FAST.

However, FAST may not achieve application launch performance close to (5) when there is a significant variation of I/O intensiveness, especially if the beginning of the launch process is more I/O intensive than the other. Fig.2 illustrates an extreme example of such a case, where the first half of this example is SSD-bound and the second half is CPU-bound. In this example, t_{CPU} is equal to t_{SSD}, and thus the expected launch time t_{expected} is given to be $t_{\mathrm{SSD}} + t(c_8)$, according to (4). However, the actual launch time t_{actual} is much larger than t_{expected}. The CPU usage in the first half of the launch time is kept quite low despite the fact that there are lots of remaining CPU computations (i.e., $c_5, \ldots, c_8$) due to the dependency between s_i and c_i. We will provide a detailed analysis for this case using real applications in Section 6.

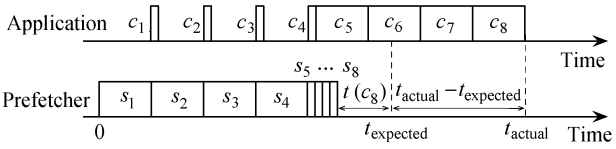


Fig.2. Worst-case example ($t_{\mathrm{CPU}} = t_{\mathrm{SSD}}$).

4 Design Consideration

We chose Linux OS to demonstrate the feasibility of FAST. Our goal is to implement FAST without modification of the OS kernel so as to lower its barrier to adoption and encourage its wide deployment. In this section, we discuss design issues that we should consider to fulfill this goal.

4.1 What to Do

Implementing FAST essentially includes the following three phases:

1) *Launch Sequence Profiling.* FAST obtains an application launch sequence for each target application.

2) *Prefetcher Generation.* FAST builds a prefetcher that replays exactly the same block I/O sequence as specified in the obtained application launch sequence.

3) *Prefetcher Execution.* FAST runs the generated prefetcher simultaneously with the original application upon detecting its launch.

For the launch sequence profiling phase, we can utilize existing block I/O tracing tools (e.g., `blktrace`[22]) to capture raw I/O request sequences from which we can extract an application launch sequence. The prefetcher execution phase can be implemented in various ways. For example, we can monitor invocation of a certain system call that launches an application program (e.g., `execve()`) to detect the launch and immediately execute the corresponding prefetcher. It is even possible to manually launch both the application and its prefetcher using a shell command. However, the prefetcher generation phase has some implementation issues — which will be discussed below — to be considered for FAST to work correctly.

4.2 I/O Replay

Since we implement an application prefetcher as a user-level program, the prefetcher of FAST should issue all I/O requests via file system calls with a file name and an offset in that file. However, the application launch sequence obtained from the launch sequence profiling phase contains only block-level information, i.e., each I/O request is represented as a tuple of the form `(start_LBA, size)`.

With these constraints in mind, we can consider the following two types of approaches to implement the prefetcher generation phase:

1) *File-Level I/O Replay.* We first convert the LBA of each I/O request into the associated file name and the file offset. We use the obtained file names and offsets to fetch the application launch sequence via file system calls.

2) *Block-Level I/O Replay.* If we open the whole block device as a file (e.g., "`/dev/sda`"), we can use the LBA of each I/O request as the file offset in the device file. With this approach, conversion from LBA to file name and offset is not necessary.

Figs. 3 and 4 show example prefetchers performing file-level and block-level I/O replay, respectively. The assumed application launch sequence is [(5, 2), (1, 1), (7, 1)] (unit: 512 B). Fig.5 shows the mapping between LBA and file offset of the example application launch sequence.

```
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;
}
```

Fig.3. Prefetcher performing file-level I/O replay.

```
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;
}
```

Fig.4. Prefetcher performing block-level I/O replay.

Although both approaches generate exactly the same sequence of I/O requests, they create different structures in the page cache due to its indexing mechanism.

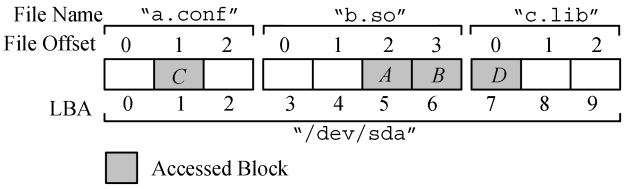


Fig.5. Launch block mapping between file offset and LBA (file offset and LBA are in unit of 512 B).

The page cache of Linux is organized as a radix tree per inode. To find a certain block in the page cache, Linux system calls pass an inode to the page cache, and only the radix tree of the designated inode is searched in the page cache.

Fig.6 shows how the two prefetchers of Figs. 3 and 4 create different page cache structures. When we perform file-level I/O replay, each fetched block is linked to the inode of its associated file (Fig.6(a)). In contrast, block-level I/O replay results in that all the cached blocks are linked to the inode of the device (Fig.6(b)).

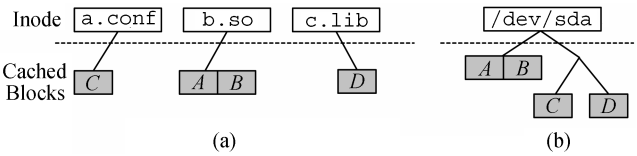


Fig.6. Page cache structures created via file- and block-level I/O replay. (a) File-level I/O replay. (b) Block-level I/O replay.

A target application issues a file system call using the file names `"a.conf"`, `"b.so"`, and `"c.lib"`. If FAST fetches the application launch sequence via block-level I/O replay, the target application cannot see the thus fetched blocks because they are linked to the radix tree of `"/dev/sda"`, as shown in Fig.6(b). In order to achieve the prefetch effect described in Subsection 3.2, FAST must create the page cache structure of Fig.6(a) via file-level I/O replay.

Hence, the implementation of FAST should include a mechanism to convert the block-level information of an application launch sequence into file-level information, i.e., LBA-to-inode reverse mapping, which will be explained in detail in Section 5.

5 Prefetcher Implementation

The implementation of FAST consists of an application launch manager, a system-call profiler, a disk I/O profiler, an application launch sequence extractor, an LBA-to-inode reverse mapper, and an application prefetcher generator.

Fig.7 shows that these components interact with each other to perform three procedures: 1) launch sequence generation, 2) LBA-to-inode map generation, and 3) prefetcher generation. The first procedure generates an application launch sequence and the second its corresponding LBA-to-inode map. The thus generated launch sequence and the LBA-to-inode map are referenced together by the third procedure to create an application prefetcher. The application launch manager monitors the launch of applications and determines when to initiate each procedure. In what follows, we detail the implementation of each component.

5.1 Launch Sequence Generation

The disk I/O profiler is used to track the block requests generated during an application launch. We used `Blktrace`[22], a built-in Linux kernel I/O-tracing tool that monitors the details of I/O behavior for the evaluation of I/O performance. `Blktrace` can profile various I/O events: inserting an item into the block layer, merging the item with a previous request in the queue, remapping onto another device, issuing a request to the device driver, and a completion signal from the device. From these events, we collect the trace of device-completion events, each of which consists of a device number, an LBA, the I/O size, and completion time.

Ideally, the application launch sequence should include all of the block requests that are generated every

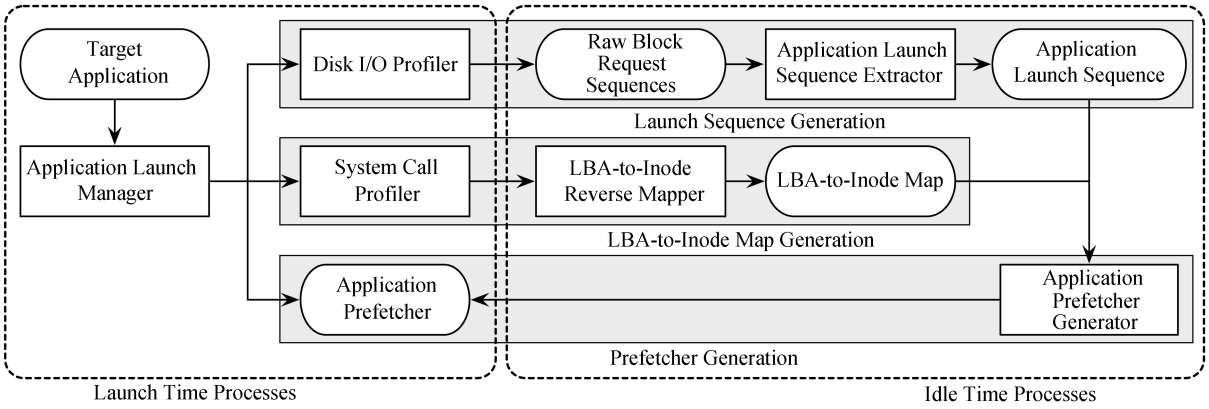


Fig.7. Structure of FAST.

time the application is launched in the cold start scenario, without including any block requests that are not relevant to the application launch. We observed that the raw block request sequence captured by `Blktrace` does not vary from one launch to another, i.e., deterministic for multiple launches of the same application. However, we observed that other processes (e.g., OS and application daemons) sometimes generate their own I/O requests simultaneously with the application launch. To handle this case, the application launch sequence extractor collects two or more raw block request sequences to extract a common sequence, which is then used as a launch sequence of the corresponding application.

Fig.8 shows the pseudo-code of the application launch sequence extractor we used. In step 1, it first removes write block requests. In step 2, it removes any block request that does not appear in all of the input sequences. This procedure makes all the input sequences the same, so we use any of them as an application launch sequence.

```
Input S_1, S_2, ...: raw block request sequences captured by
      the disk I/O profiler
Output S = (s_1, ..., s_n): an application launch sequence
begin
  for each S_i do
    for each s_j of S_i do
      if s_j is a write request // step 1
        then remove s_j from S_i
      fi;
      if there is any S_k not including s_j // step 2
        then remove s_j from all S_i's
      fi;
    od;
  od;
  Return any of S_i's
end
```

Fig.8. Application launch sequence extractor.

5.2 LBA-to-Inode Map Generation

Most file systems, including EXT3, do not support a reverse mapping from LBA to file name and offset. However, for a given file name, we can easily find the LBA of all of the blocks that belong to the file and their offsets in the file. Hence, we can build an LBA-to-inode map by gathering this information for every file. However, building such a map of the entire file system is impractical because a file system, in general, contains tens of thousands of files and their block locations on the disk change very often.

Therefore, we build a separate LBA-to-inode map for each application, which can significantly reduce the overhead of creating an LBA-to-inode map because 1) the number of applications and the number of files used in launching each application are very small compared to the number of files in the entire file system, and 2) most of them are shared libraries and application code blocks, so their block locations remain unchanged unless they are updated or disk defragmentation is performed.

We implement the LBA-to-inode reverse mapper that receives a list of file names as input and creates an LBA-to-inode map as output. An LBA-to-inode map is built using a red-black tree in order to reduce the search time. Each node in the red-black tree has the LBA of a block as its key, and a block type as its data by default. According to the block type, different types of data are added to the node. A block type includes a super block, a group descriptor, an inode block bitmap, a data block bitmap, an inode table, and a data block. For example, a node for a data block has a block type, a device number, an inode number, an offset, and a size. Also, for a data block, a table is created to keep the mapping information between an inode number and its file name.

The system-call profiler obtains a full list of file names that are accessed during an application launch, and passes it to the LBA-to-inode reverse mapper. We used `strace` for the system-call profiler, which is a debugging tool in Linux. We can specify the argument of `strace` so that it may monitor only the system calls that have a file name as their argument. As many of these system calls are rarely called during an application launch, we monitor only the following system calls that frequently occur during application launches: `open()`, `creat()`, `execve()`, `stat()`, `stat64()`, `lstat()`, `lstat64()`, `access()`, `truncate()`, `truncate64()`, `statfs()`, `statfs64()`, `readlink()`, and `unlink()`.

5.3 Prefetcher Generation

The application prefetcher is a user-level program that replays the disk access requests made by a target application. We implemented the application prefetcher generator to automatically create an application prefetcher for each target application. It performs the following operations.

1) Read s_i one-by-one from S of the target application.

2) Convert s_i into its associated data items stored in the LBA-to-inode map, e.g., `(dev,LBA,size)`→`(datablk,filename,offset,size)` or `(dev,LBA,size)`→`(inode,start_inode,end_inode)`.

3) Depending on the type of block, generate an appropriate system call using the converted disk access information.

4) Repeat steps 1∼3 until processing all s_i.

Table 1 shows the kind of system calls used for each block type. There are two system calls that can be used to replay the disk access for data blocks of a regular file. If we use `read()`, data is first moved from the SSD to the page cache, and then copying takes place from the page cache to the user buffer. The second step is unnecessary for our purpose, as the process that actually manipulates the data is not the application prefetcher but the target application. Hence, we chose `posix_fadvise()` that performs only the first step, from which we can avoid the overhead of `read()`. We used the `POSIX_FADV_WILLNEED` parameter, which informs the OS that the specified data will be used in the near future. When to issue the corresponding disk access after `posix_fadvise()` is called depends on the OS implementation. We confirmed that the current version of Linux we used issues a block request immediately after receiving the information through `posix_fadvise()`, thus meeting our need. A symbolic-linked file name is stored in data block pointers in an inode entry when the length of the file name is less than or equal to 60 B (c.f., the space of data block pointers is 60 B, 4×12 for direct, 4 for single indirect, another 4 for double indirect, and last 4 for triple indirect data block pointer). If the length of linked file name is more than 60 B, the name is stored in the data blocks pointed to by data block pointers in the inode entry. We used `readlink()` to replay the data block access of symbolic-link file names that are longer than 60 B.

Table 1. System Calls to Replay Block I/O Requests

Block Type	System Call
Inode table	`open()`
Data block: A directory	`opendir()` and `readdir()`
Data block: A regular file	`read()` or `posix_fadvise()`
Data block: A symbolic link file	`readlink()`

Fig.9 is an example of automatically-generated application prefetcher. Unlike the target application, the application prefetcher successively fetches all the blocks as soon as possible to minimize the time between adjacent block requests.

In the EXT3 file system, the inode of a file includes pointers of up to 12 data blocks, so these blocks can be found immediately after accessing the inode. If the file size exceeds 12 blocks, indirect, double indirect, and triple indirect pointer blocks are used to store the pointers to the data blocks. Therefore, requests for indirect pointer blocks may occur in the cold start scenario when the application is accessing files larger than 12 blocks. We cannot explicitly load those indirect pointer blocks in the application prefetcher because there is no such system call. However, the `posix_fadvise()` call for a data block will first make a request for the indirect block when needed, so it can be fetched in a timely manner by running the application prefetcher.

```
int main(void) {
 ...
 readlink("/etc/fonts/conf.d/90-ttf-arphic-uming-embolde
 n.conf", linkbuf, 256);
 int fd423;
 fd423 = open("/etc/fonts/conf.d/90-ttf-arphic-uming-emb
 olden.conf", O_RDONLY);
 posix_fadvise(fd423, 0, 4096, POSIX_FADV_WILLNEED);
 posix_fadvise(fd351, 286720, 114688, POSIX_FADV_WILLNEE
 D);
 int fd424;
 fd424 = open("/usr/share/fontconfig/conf.avail/90-ttf-a
 rphic-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;
}
```

Fig.9. Example application prefetcher.

The following types of block request are not listed in Table 1: a superblock, a group descriptor, an inode entry bitmap, a data block bitmap. We found that requests to these types of blocks seldom occur during an application launch, so we did not consider their prefetching.

5.4 Application Launch Manager

The role of the application launch manager is to detect the launch of an application and to take an appropriate action. We can detect the beginning of an application launch by monitoring `execve()` system call, which is implemented using a system-call wrapper. There are three phases with which the application launch manager deals: a launch sequence profiling phase, a prefetcher generation phase, and a prefetcher execution phase. The application launch manager uses a set of variables and parameters for each application to decide when to change its phase. These are summarized in Table 2.

Here we describe the operations performed in each phase:

1) *Launch Sequence Profiling.* If no application prefetcher is found for that application, the application launch manager regards the current launch as the first launch of this application, and enters the initial launch phase. In this phase, the application launch manager performs the following operations in addition to the launch of the target application:

- Increase n_{init} of the current application by 1.
- If $n_{\text{init}} = 1$, run the system call profiler.

Table 2. Variables and Parameters Used by the Application Launch Manager

Type	Description
n_{init}	A counter to record the number of application launches done in the initial launch phase
n_{pref}	A counter to record the number of launches done in the application prefetch phase after the last check of the miss ratio of the application prefetcher
N_{rawseq}	The number of raw block request sequences that are to be captured at the launch profiling phase
N_{chk}	The period to check the miss ratio of the application prefetcher
R_{miss}	A threshold value for the prefetcher miss ratio that is used to determine if an update of the application or shared libraries has taken place
T_{idle}	A threshold value for the idle time period that is used to determine if an application launch is completed
T_{timeout}	The maximum amount of time allowed for the disk I/O profiler to capture block requests

• Flush the page cache, dentries (directory entries), and inodes in the main memory to ensure a cold start scenario, which is done by the following command:

```
$ echo 3 > /proc/sys/vm/drop_caches.
```

• Run the disk I/O profiler. Terminate the disk I/O profiler when any of the following conditions are met: if no block request occurs during the last T_{idle} seconds or the elapsed time since the start of the disk I/O profiler exceeds T_{timeout} seconds.

• If $n_{\text{init}} = N_{\text{rawseq}}$, enter the prefetcher generation phase after the current launch is completed.

2) *Prefetcher Generation.* Once application launch profiling is done, it is ready to generate an application prefetcher using the information obtained from the first phase. This can be performed either immediately after the application launch is completed, or when the system is idle. The following operations are performed:

• Run the application launch sequence extractor.

• Run the LBA-to-inode reverse mapper.

• Run the application prefetcher generator.

• Reset the values of n_{init} and n_{pref} to 0.

3) *Prefetcher Execution.* If the application prefetcher for the current application is found, the application launch manager runs the prefetcher simultaneously with the target application. It also periodically checks the miss ratio of the prefetcher to determine if there has been any update of the application or shared libraries. Specifically, the following operations are performed:

• Increase n_{pref} of the current application by 1.

• If $n_{\text{pref}} = N_{\text{chk}}$, reset the value of n_{pref} to 0 and run the disk I/O profiler. Its termination conditions are the same as those in the first phase.

• Run the application prefetcher simultaneously with the target application.

• If a raw block request sequence is captured, use it to calculate the miss ratio of the application prefetcher. If it exceeds R_{miss}, delete the application prefetcher.

The miss ratio is defined as the ratio of the number of block requests not issued by the prefetcher to the total number of block requests in the application launch sequence.

6 Performance Evaluation

In this section, we evaluate the performance of FAST using various applications and discuss a set of implementation issues that can affect the efficiency of FAST.

6.1 Experimental Setup

6.1.1 Experimental Platform

We used a desktop PC equipped with an Intel i7-860 2.8 GHz CPU, 4 GB of PC12800 DDR3 SDRAM and an Intel 80 GB SSD (X25-M G2 Mainstream). We installed a Fedora 12 with Linux kernel 2.6.32 on the desktop, in which we set NOOP as the default I/O scheduler. For benchmark applications, we chose frequently used user-interactive applications, for which application launch performance matters much. Such an application typically uses graphical user interfaces and requires user interaction immediately after completing its launch. Applications like gcc and gzip are not included in our set of benchmarks as launch performance is not an issue for them. Our benchmark set consists of the following Linux applications: Acrobat Reader, Designer-qt4, Eclipse, F-Spot, Firefox, Gimp, Gnome, Houdini, Kdevdesigner, Kdevelop, Konqueror, Labview, Matlab, OpenOffice, Skype, Thunderbird, and Xilinx ISE. In addition to these, we used Wine[23], which is an implementation of the Windows API running on Linux OS, to test Access, Excel, PowerPoint, Visio, and Word — typical Windows applications.

6.1.2 Test Scenarios

For each benchmark application, we measured its launch time for the following scenarios.

• *Cold Start.* The application was launched immediately after flushing the page cache, using the method described in Subsection 5.4. The resulting launch time is denoted by t_{cold}.

• *Warm Start.* We first ran the application prefetcher only to load all the blocks in the application launch sequence to the page cache, and then launched the application. Let t_{warm} denote the resulting launch time.

• *Sorted Prefetch.* To evaluate the performance of the sorted prefetch[12–14] on SSDs, we modified the application prefetcher to fetch the block requests in the application launch sequence in the sorted order of their

LBAs. After flushing the page cache, we first ran the modified application prefetcher, then immediately ran the application. Let t_{sorted} denote the resulting launch time.

• *FAST.* We flushed the page cache, and then ran the application simultaneously with the application prefetcher. The resulting launch time is denoted by t_{FAST}.

• *Prefetcher Only.* We flushed the page cache and ran the application prefetcher. The completion time of the application prefetcher is denoted by t_{SSD}. It is used to calculate a lower bound of the application launch time $t_{\text{bound}} = \max(t_{\text{SSD}}, t_{\text{CPU}})$, where $t_{\text{CPU}} = t_{\text{warm}}$ is assumed.

6.1.3 Launch-Time Measurement

We start an application launch by clicking an icon or inputting a command, and can accurately measure the launch start time by monitoring when `execve()` is called. Although it is difficult to clearly define the completion of a launch, a reasonable definition is the first moment the application becomes responsive to the user[4]. However, it is difficult to accurately and automatically measure that moment. So, as an alternative, we measured the completion time of the last block request in an application launch sequence using `Blktrace`, assuming that the launch would be completed very soon after issuing the last block request. For the warm start scenario, we executed `posix_fadvise()` with `POSIX_FADV_DONTNEED` parameter to evict the last block request from the page cache. For the sorted prefetch and the FAST scenarios, we modified the application prefetcher so that it skips prefetching of the last block request.

6.2 Experimental Results

6.2.1 Application Launch Sequence Generation

We captured 10 raw block request sequences during the cold start launch of each application. We ran the application launch sequence extractor with a various number of input block request sequences, and observed the size of the resulting application launch sequences.

Fig.10 shows that for all the applications we tested, there is no significant reduction of the application launch sequence size while increasing the number of inputs from 2 to 10. Hence, we set the value of N_{rawseq} in Table 2 to 2 in this paper. We used the size of the first captured input sequence as the number of input one in Fig.10 (the application launch sequence extractor requires at least two input sequences). For some applications, there are noticeable differences in size between the number of inputs one and two. This is because the first raw input request sequence includes a set of bursty I/O requests generated by OS and user daemons that are irrelevant to the application launch. Fig.10 shows that such I/O requests can be effectively excluded from the resulting application launch sequence using just two input request sequences.

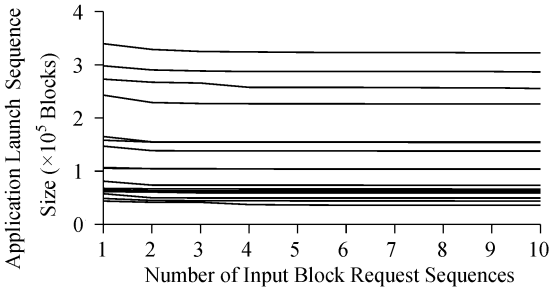


Fig.10. Size of application launch sequences.

The second and third columns of Table 3 summarize the total number of block requests and accessed blocks of the thus-obtained application launch sequences, respectively. The last column shows the total number of files used during the launch of each application.

Table 3. Collected Launch Sequences

Application	Number of Block Requests	Number of Fetched Blocks	Number of Used Files
Access	1 296	106 992	555
Acrobat Reader	960	73 784	178
Designer-qt4	2 400	138 608	410
Eclipse	4 163	155 216	787
Excel	1 610	169 112	583
F-Spot	1 180	49 968	304
Firefox	1 566	60 944	433
Gimp	1 939	66 928	799
Gnome	4 739	228 872	538
Houdini	4 836	290 320	724
Kdevdesigner	1 537	44 904	467
Kdevelop	1 970	63 104	372
Konqueror	1 780	62 216	296
Labview	2 927	154 768	354
Matlab	6 125	267 312	742
OpenOffice	1 425	104 600	308
PowerPoint	1 405	120 808	576
Skype	892	41 560	197
Thunderbird	1 533	64 784	429
Visio	1 769	168 832	662
Word	1 715	181 496	613
Xilinx ISE	4 718	328 768	351

Note: $N_{\text{rawseq}} = 2$.

6.2.2 Testing of the Application Prefetcher

Application prefetchers are automatically generated for the benchmark applications using the application launch sequences in Table 3. In order to see if the application prefetchers fetch all the blocks used by an application, we first flushed the page cache, and launched

each application immediately after running the application prefetcher. During the application launch, we captured all the block requests generated using `Blktrace`, and counted the number of missed block requests. The average number of missed block requests was 1.6% of the number of block requests in the application launch sequence, but varied among repeated launches, e.g., from 0% to 6.1% in the experiments we performed.

By examining the missed block requests, we could categorize them into three types: 1) files opened by OS daemons and user daemons at boot time; 2) journaling data or swap partition accesses; and 3) files dynamically created or renamed at every launch (e.g., `tmpfile()`). The first type occurs because we force the page cache to be flushed in the experiment. In reality, they are highly likely to reside in the page cache, and thus, this type of misses will not be a problem. The second type is irrelevant to the application, and observed even during idle time. The third type occurs more or less often, depending on the application. FAST does not prefetch this type of block requests as they change at every launch.

6.2.3 Experiments for the Test Scenarios

We measured the launch time of the benchmark applications for each test scenario listed in Subsection 6.1. Fig.11 shows that the average launch time reduction of FAST is 28% over the cold start scenario. The performance of FAST varies considerably among applications, ranging from 16% to 46% reduction of launch time. In particular, FAST shows performance very close to t_{bound} for some applications, such as Eclipse, Gnome, and Houdini. On the other hand, the gap between t_{bound} and t_{FAST} is relatively larger for such applications as Acrobat Reader, Firefox, OpenOffice, and Labview.

6.2.4 Launch Time Behavior

We conducted experiments to see if the application prefetcher works well as expected when it is simultaneously run with the application. We chose Firefox because it shows a large gap between t_{bound} and t_{FAST}. We monitored the generated block requests during the launch of Firefox with the application prefetcher, and observed that the first 12 of the entire 1 566 block requests were issued by Firefox, which took about 15 ms. As the application prefetcher itself should be launched as well, FAST cannot prefetch these block requests until finishing its launch. However, we observed that all the remaining block requests were issued by FAST, meaning that they were successfully prefetched before the CPU needed them.

6.2.5 CPU and SSD Usage Patterns

We performed another experiment to observe the CPU and SSD usage patterns in each test scenario. We chose two applications, Eclipse and Firefox, representing the two groups of applications of which t_{FAST} is close to and far from t_{bound}, respectively. We modified the OS kernel to sample the number of CPU cores having runnable processes and to count the number of cores in the I/O wait state. Fig.12 shows the CPU and SSD usage of the two applications, where the entire CPU is regarded as busy if at least one of its cores is active. Similarly, the SSD is assumed busy if there are one or more cores in the I/O wait state. In the cold start scenario, there is almost no overlap between CPU computation and SSD access for both applications. In the warm start scenario, the CPU stays fully active until the launch is completed as there is no wait. One exception we observed is the time period marked with circle (a), during which the CPU seemed to be in the event-waiting state. FAST is shown to be successful in overlapping CPU computation with SSD access as we intended. However, CPU usage is observed to be low at the beginning of launch for both applications, which can be explained with the example in Fig.2. As Eclipse shows a shorter such time period (circle (b)) than Firefox (circle (c)), t_{FAST} can reach closer to t_{bound}. In the case of Firefox, however, the ratio of t_{CPU} to t_{SSD} is close to 1:1, allowing FAST to achieve more reduction of launch time for Firefox than for Eclipse.

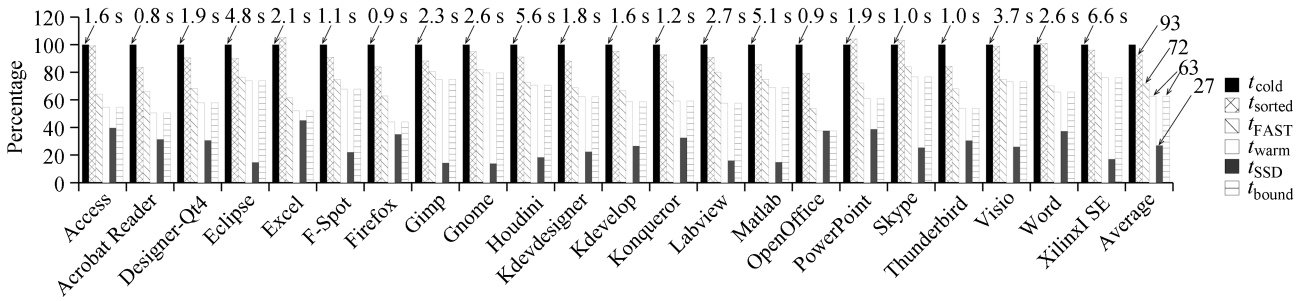


Fig.11. Measured application launch time (normalized to t_{cold}).

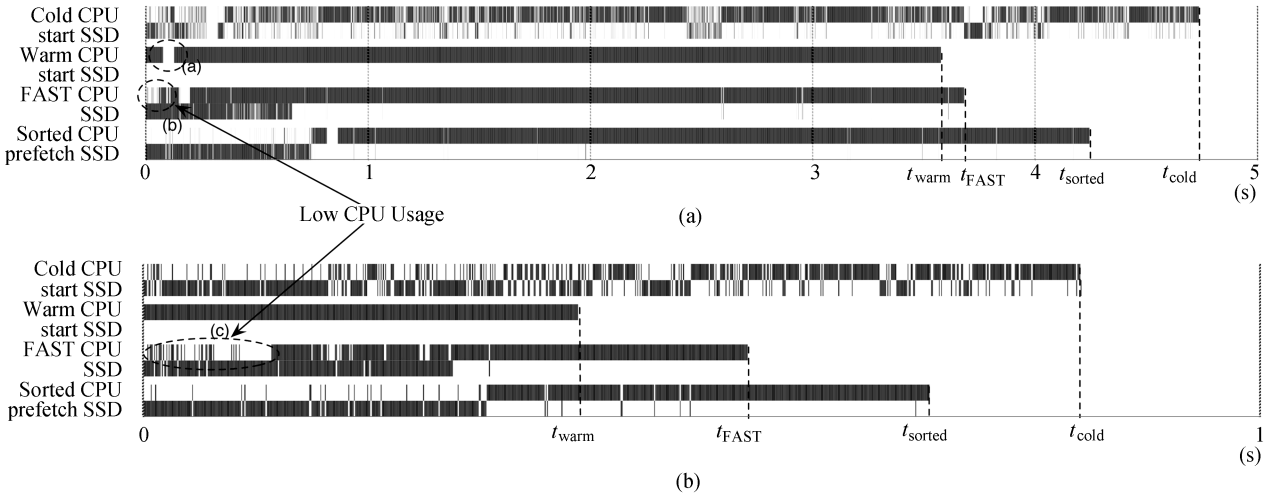


Fig.12. Usage of CPU and SSD (sampling rate = 1 KHz). (a) Application: Eclipse. (b) Application: Firefox.

6.2.6 *Performance of Sorted Prefetch*

Fig.11 shows that the sorted prefetch reduces the application launch time by an average of 7%, which is less efficient than FAST, but non-negligible. One reason for this improvement is the difference in I/O burstiness between the cold start and the sorted prefetch. Most SSDs (including the one we used) support the native command queueing (NCQ) feature, which allows up to 31 block requests to be sent to an SSD controller. Using this information, the SSD controller can read as many NAND flash chips as possible, effectively increasing read throughput. The average queue depth in the cold start scenario is close to 1, meaning that for most of time there is only one outstanding request in case of SSD. In contrast, in the sorted prefetch scenario, the queue depth will likely grow larger than 1 because the prefetcher may successively issue asynchronous I/O requests using `posix_fadvise()`, at small inter-issue intervals.

On the other hand, we could not find a clear evidence that sorting block requests in their LBA order is advantageous in case of SSD. Rather, the execution time of the sorted prefetcher was slightly longer than its unsorted version for most of the applications we tested. Also, the sorted prefetch shows worse performance than the cold start for Excel, PowerPoint, Skype, and Word. Although these observations were consistent over repeated tests, a further investigation is necessary to understand such a behavior.

6.3 Implementation Issues

6.3.1 *Simultaneous Launch of Applications*

We performed experiments to see how well FAST can scale up for launching multiple applications. We launched multiple applications starting from the top of Table 3, adding 5 at a time, and measured the launch completion time of all launched applications[①]. Fig.13 shows that FAST could reduce the launch completion time for all the tests, whereas the sorted prefetch did not scale beyond 10 applications. Note that the FAST improvement decreased from 20% to 7% as the number of applications increased from 5 to 20.

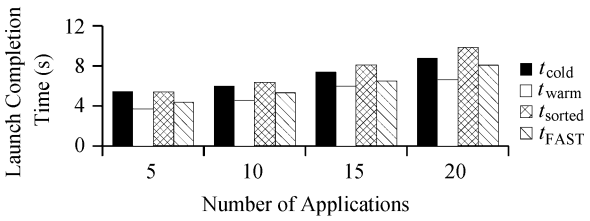


Fig.13. Simultaneous launch of multiple applications.

6.3.2 *Runtime and Space Overhead*

We analyzed the runtime overhead of FAST for seven possible combinations of running processes, and summarized the results in Table 4. Cases 2 and 3 belong to the *launch profiling* phase, which was described in Subsection 5.4. During this phase, Case 2 occurs only

Table 4. Runtime Overhead

Running Processes	Runtime (s)
1. Application Only (cold start scenario)	0.86
2. `strace` + `blktrace` + Application	1.21
3. `blktrace` + Application	0.88
4. Prefetcher Generation	5.01
5. Prefetcher + Application	0.56
6. Prefetcher + `blktrace` + Application	0.59
7. Miss Ratio Calculation	0.90

Note: Application: Firefox.

[①]Except for Gnome that cannot be launched with other applications, and Houdini whose license had expired.

once, and Case 3 occurs N_{rawseq} times. Case 4 corresponds to the *prefetcher generation* phase (the right side of Fig.7), and shows a relatively long runtime. However, we can hide it from users by running it in background. Also, since we primarily focused on functionality in the current implementation, there is room for further optimization. Cases 5, 6, and 7 belong to the *application prefetch* phase, and repeatedly occur until the application prefetcher is invalidated. Cases 6 and 7 occur only when n_{pref} reaches N_{chk}, and Case 7 can be run in background.

FAST creates temporary files such as system call log files and I/O traces, but these can be deleted after FAST completes creating application prefetchers. However, the generated prefetchers occupy disk space as far as application prefetching is used. In addition, application launch sequences are stored to check the miss ratio of the corresponding application prefetcher. In our experiment, the total size of the application prefetchers and application launch sequences for all 22 applications was 7.2 MB.

6.3.3 Scenarios Making FAST Inefficient

While previous examples clearly demonstrate the benefits of FAST for a wide range of applications, FAST does not guarantee improvements for all cases. One such a scenario is when a target application is too small to offset the overhead of loading the prefetcher. We tested FAST with the Linux utility `uname`, which displays the name of the OS. It generated three I/O requests whose total size was 32 KB. The measured t_{cold} was 2.2 ms, and t_{FAST} was 2.3 ms, 5% longer than the cold start time.

Another possible scenario is when the target application experiences a major update. In this scenario, FAST may fetch data that will not be used by the newly updated application until it detects the application update and enters a new launch profiling phase. We modified the application prefetcher so that it fetches the same size of data from the same file but from another offset that is not used by the application. We tested the modified prefetcher with Firefox. Even in this case, FAST reduced application launch time by 4%, because FAST could still prefetch some of the metadata used by the application. Assuming most of the file names are changed after the update, we ran Firefox with the prefetcher for Gimp, which fetches a similar number of blocks as Firefox. In this experiment, the measured application launch time was 7% longer than the cold start time, but the performance degradation was not drastic due to the internal parallelism of the SSD we used (10 channels).

6.3.4 Application Launch Sequence Determinism on Multi-Core CPUs

The CPU we used in the experiment has four cores and is able to simultaneously execute up to 8 threads through the Intel *HyperThreading* technology. The use of a multi-core CPU can affect the launch sequence determinism because an application can dynamically create multiple threads during its launch process.

However, the experimental results in Subsection 6.2 demonstrate that all of the tested benchmark applications show launch sequence determinism even on the multi-core CPU, allowing FAST to effectively reduce application launch time by exploiting the determinism.

For detailed analysis, we visualized the number of CPUs that were active during the cold start scenario launch process of Eclipse application in Fig.14, where we set the sampling frequency to 1 KHz and took total 4 738 samples. The two main observations we made from Fig.14 are as below:

1) The SSD is mostly idle when the CPU is active and vice versa. Among the whole 4 738 samples, the CPU and the SSD are both active only in 109 samples (2%).

2) When the CPU is active, mostly only one CPU core is used. Among the 3 685 samples where one or more CPU cores are active, 3 623 samples (or 98%) have only one active CPU core.

To summarize, there is not much parallelism among the threads that are created during the launch process.

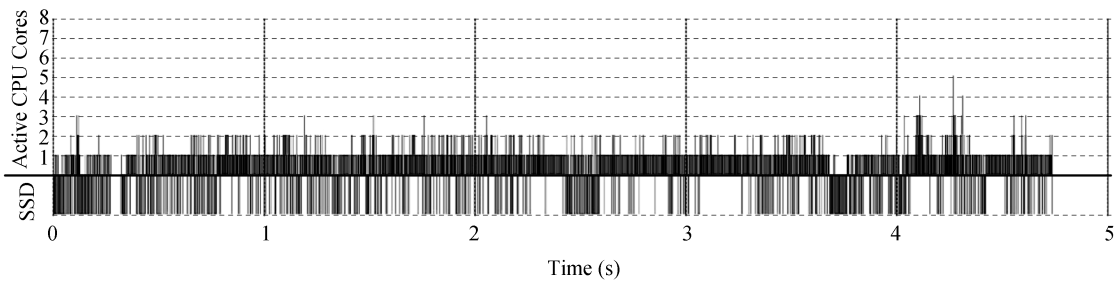


Fig.14. CPU and SSD usage during an application launch process. (CPU: Intel i7-860 2.8 GHz, SSD: 80 GB Intel X25-M G2 Mainstream, application: Eclipse.)

Consequently, the launch sequence determinism can be maintained even on the multi-core CPU systems.

6.3.5 *Configuring Application Launch Manager*

The application launch manager has a set of parameters to be configured, as shown in Table 2. If N_{rawseq} is set too large, users will experience the cold-start performance during the initialization phase. If it is set too small, unnecessary blocks may be included in the application prefetcher. Fig.10 shows that setting it between 2 and 4 is a good choice. The proper value of N_{chk} will depend on the runtime overhead of `Blktrace`; if FAST is placed in the OS kernel, the miss ratio of the application prefetcher may be checked upon every launch ($N_{\text{chk}} = 1$) without noticeable overhead. Also, setting R_{miss} to 0.1 is reasonable, but it needs to be adjusted after gaining enough experience in using FAST. To find the proper value of T_{idle}, we investigated the SSD's maximum idle time during the cold-start of applications, and found it to range from 24 ms (Thunderbird) to 826 ms (Xilinx ISE). Hence, setting T_{idle} to 2 seconds is proper in practice. As the maximum cold-start launch time is observed to be less than 10 seconds, 30 seconds may be reasonable for T_{timeout}. All these values may need to be adjusted, depending on the underlying OS and applications.

7 Applicability of FAST

FAST can be used not only for SSDs but also for other types of storage devices. Here we discuss the applicability of FAST on HDDs and smartphones.

7.1 FAST on HDDs

To see how FAST works on an HDD, we replaced the SSD with a Seagate 3.5″ 1 TB HDD (ST31000528AS) and measured the launch time of the same set of benchmark applications, of which the result is shown in Fig.15. Although FAST worked well as expected by hiding most of CPU computation from the application launch, the average launch time reduction was only 15%. It is because the application launch on an HDD is mostly I/O bound; in the cold start scenario, we observed that about 85% of the application launch time was spent on accessing the HDD. In contrast, the sorted prefetch was shown to be more effective; it could reduce the application launch time by an average of 40% by optimizing disk head movements.

We performed another experiment by modifying the sorted prefetch so that the prefetcher starts simultaneously with the original application, like FAST. However, the resulting launch time reduction was only 19% (denoted by t_{mFAST} in Fig.15), which is worse than that of the unmodified sorted prefetch. The performance degradation is due to the I/O contention between the prefetcher and the application.

7.2 FAST on Smartphones

The similarity between modern smartphones and PCs with SSDs in terms of the internal structure and the usage pattern, as summarized below, makes smartphones a good candidate to which we can apply FAST.

- Unlike other mobile embedded systems, smartphones run different applications at different times, making application launch performance matter more;
- Smartphones use NAND flash as their secondary storage, of which the performance characteristics are basically the same as the SSD; and
- Smartphones often use slightly customized (if not the same) OSes and file systems that are designed for PCs, reducing the effort to port FAST to smartphones.

Furthermore, a smartphone has the characteristics that enhance the benefit of using FAST as follows:

- Users tend to launch and quit applications more frequently on smartphones than on PCs;
- Due to relatively smaller main memory of a smartphone, users will experience cold start performance more frequently; and
- Its relatively slower CPU and flash storage speed may increase the absolute reduction of application launch time by applying FAST.

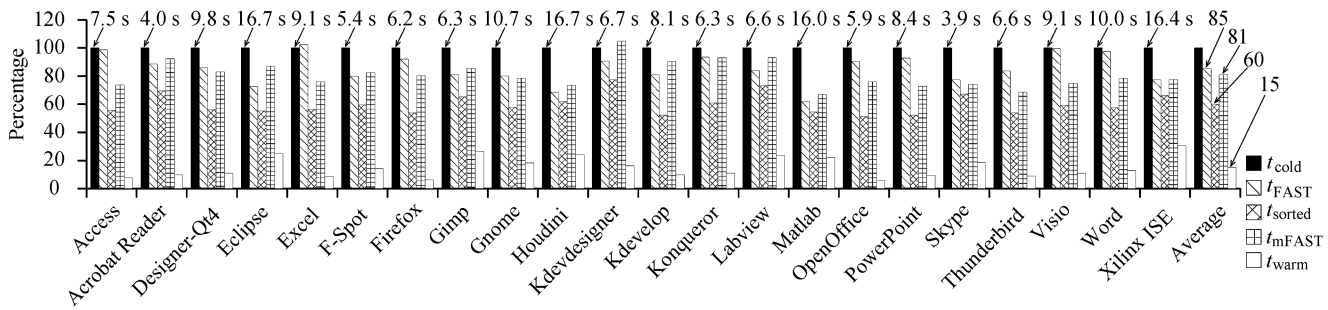


Fig.15. Measured application launch time on an HDD (normalized to t_{cold}).

Although we have not yet implemented FAST on a smartphone, we could measure the launch time of some smartphone applications by simply using a stopwatch. We randomly chose 14 applications installed on the iPhone 4 to compare their cold and warm start time, of which the results are plotted in Fig.16. The average cold start time of the smartphone applications is 6.1 seconds, which is more than twice of the average cold start time of the PC applications (2.4 seconds) shown in Fig.11. Fig.16 also shows that the average warm start time is 63% of the cold start time (almost the same ratio as in Fig.11), implying that we can achieve similar benefits from applying FAST to smartphones.

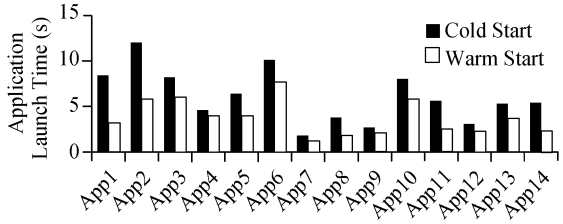


Fig.16. Measured application launch time on iPhone 4 (CPU: 1 GHz, SDRAM: 512 MB, NAND flash: 32 GB).

8 Comparison of FAST with Traditional Prefetching

FAST is a special type of prefetching optimized for application launch, whereas most of the traditional prefetching schemes focus on runtime performance improvement. We compare FAST with the traditional prefetching algorithms by answering the following three questions inspired by previous work[24].

8.1 What to Prefetch

FAST prefetches the blocks appear in the application launch sequence. While many prediction-based prefetching schemes[25-27] suffer from low hit ratio of the prefetched data, FAST can achieve near 100% hit ratio. This is because the application launch sequence changes little over repeated launches of an application.

Sequential pattern detection schemes like readahead[28-29] can achieve a fairly good hit ratio when activated, but they are applicable only when such a pattern is detected. By contrast, FAST guarantees stable performance for every application launch.

One way to enhance the prefetch hit ratio for a complicated disk I/O pattern is to analyze the application source code to extract its access pattern. Using the thus-obtained pattern, prefetching can be done by either inserting prefetch codes into the application source code[30-31] or converting the source code into a computation thread and a prefetch thread[32]. However, such an approach does not work well for application launch optimization because many of the block requests generated during an application launch are not from the application itself but from other sources, such as loading shared libraries, which cannot be analyzed by examining the application source code. Furthermore, both require modification of the source code, which is usually not available for most commercial applications. Even if the source code is available, modifying and recompiling every application would be tedious and inconvenient. In contrast, FAST does not require application source code and is thus applicable for any commercial application.

Another relevant approach[33] is to deploy a shadow process that speculatively executes the copy of the original application to get hints for the future I/O requests. It does not require any source modification, but consumes non-negligible CPU and memory resources for the shadow process. Although it is acceptable when CPU is otherwise stalled waiting for the I/O completion, employing such a shadow process in FAST may degrade application launch performance as there is not enough CPU idle period as shown in Fig.12.

8.2 When to Prefetch

FAST is not activated until an application is launched, which is as conservative as demand paging. Thus, unlike prediction-based application prefetching schemes[10-11], there is no cache-pollution problem or additional disk I/O activity during idle period. However, once activated, FAST aggressively performs prefetching: it keeps on fetching subsequent blocks in the application launch sequence asynchronously even in the absence of page misses. As the prefetched blocks are mostly (if not all) used by the application, the performance improvement of FAST is comparable to that of the prediction-based schemes when their prediction is accurate.

8.3 What to Replace

FAST does not modify the replacement algorithm of page cache in main memory, so the default page replacement algorithm is used to determine which page to evict in order to secure free space for the prefetched blocks.

In general, prefetching may significantly affect the performance of page replacement. Thus, previous work[34-36] emphasized the need for integrated prefetching and caching. However, FAST differs from the traditional prefetching schemes since it prefetches only those blocks that will be referenced before the application launch completes (e.g., in next few seconds). If the page cache in the main memory is large enough to store all

the blocks in the application launch sequence, which is commonly the case, FAST will have minimal effect on the optimality of the page replacement algorithm.

9 Conclusions

We proposed a new I/O prefetching technique called FAST for the reduction of application launch time on SSDs. We implemented and evaluated FAST on the Linux OS, demonstrating its deployability and performance superiority. While the HDD-aware application launcher showed only 7% of launch time reduction on SSDs, FAST achieved a 28% reduction with no additional overhead, demonstrating the need for, and the utility of, a new SSD-aware optimizer. FAST with a well-designed entry-level SSD can provide end-users the fastest application launch performance. It also incurs fairly low implementation overhead and has excellent portability, facilitating its wide deployment in various platforms.

References

[1] Larus J. Spending Moore's dividend. *Commun. ACM*, 2009, 52(5): 62-69.

[2] Microsoft. Support and Q&A for solid-state drives. http://blogs.msdn.com/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx, May 2009.

[3] Dunn M, Reddy A L N. A new I/O scheduler for solid state devices. Technical Report TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.

[4] Apple Inc. Launch time performance guidelines. http://developer.apple.com/documentation/Performance/Conceptual/LaunchTime/LaunchTime.pdf, April 2006.

[5] Nadgir N. Reducing application startup time in the Solaris 8 OS. http://developers.sun.com/solaris/articles/reducing_app.html, January 2002.

[6] Colitti L. Analyzing and improving GNOME startup time. In *Proc. the 5th System Administration and Network Engineering Conference*, May 2006, pp.1-11.

[7] Jo H, Kim H, Jeong J, Lee J, Maeng S. Optimizing the startup time of embedded systems: A case study of digital TV. *IEEE Trans. Consumer Electron.*, 2009, 55(4): 2242-2247.

[8] Park C, Kim K, Jang Y, Hyun K. Linux bootup time reduction for digital still camera. In *Proc. the Linux Symposium*, July 2006, pp.239-248.

[9] Kaminaga H. Improving Linux startup time using software resume. In *Proc. the Linux Symposium*, July 2006, pp.25-34.

[10] Microsoft. Windows PC accelerators. http://www.microsoft.com/whdc/system/sysperf/perfaccel.mspx, May 2012.

[11] Esfahbod B. Preload — An adaptive prefetching daemon [Master's Thesis]. Graduate Department of Computer Science, University of Toronto, Canada, 2006.

[12] Russinovich M E, Solomon D. Microsoft Windows Internals. Microsoft Press (4th edition), December 2004, pap.458-462.

[13] Hubert B. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. In *Proc. the Linux Symposium*, July 2005, pp.245-248.

[14] Prefetch: Linux solution for prefetching necessary data during application and system startup. http://code.google.com/p/prefetch/, August 2007.

[15] Chen F, Koufaty D A, Zhang X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. the 11th ACM SIGMETRICS International Joint Conference on Measurement and Modeling of Computer Systems*, June 2009, pp.181-192.

[16] Dirik C, Jacob B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proc. the 36th International Symposium on Computer Architecture*, June 2009, pp.279-289.

[17] Kim H, Ahn S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proc. the 6th USENIX Conference on File and Storage Technologies*, February 2008, Article No.16.

[18] Kim J, Oh Y, Kim E, Choi J, Lee D, Noh S H. Disk schedulers for solid state drivers. In *Proc. the 7th ACM International Conference on Embedded Software*, October 2009, pp.295-304.

[19] Bhadkamkar M, Guerra J, Useche L, Burnett S, Liptak J, Rangaswami R, Hristidis V. BORG: Block-reORGanization for self-optimizing storage systems. In *Proc. the USENIX Conference on File and Storage Technologies*, February 2009, pp.183-196.

[20] Li Z, Chen Z, Srinivasan S M, Zhou Y. C-Miner: Mining block correlations in storage systems. In *Proc. the 3rd USENIX Conference on File and Storage Technologies*, March 31-April 2, 2004, pp.173-186.

[21] Hsu W W, Smith A J, Young H C. The automatic improvement of locality in storage systems. *ACM Trans. Comput. Syst.*, 2005, 23(4): 424-473.

[22] Axboe J. Block IO tracing. http://www.kernel.org/git/?p=linux/kernel/git/axboe/blktrace.git;a=blob;f=README, September 2006.

[23] Wine User Guide. http://www.winehq.org/docs/wineusrguide/index, 8 May 2012.

[24] Papathanasiou A E, Scott M L. Energy efficient prefetching and caching. In *Proc. the USENIX Annual Technical Conference*, June 2004, p.22.

[25] Curewitz K M, Krishnan P, Vitter J S. Practical prefetching via data compression. *SIGMOD Rec.*, 1993, 22(2): 257-266.

[26] Kotz D, Ellis C S. Practical prefetching techniques for parallel file systems. In *Proc. the 1st International Conference on Parallel and Distributed Information Systems*, December 1991, pp.182-189.

[27] Vellanki V, Chervenak A L. A cost-benefit scheme for high performance predictive prefetching. In *Proc. the ACM/IEEE Conference on Supercomputing*, November 1999, Article No. 50.

[28] Wu F, Xi H, Xu C. On the design of a new Linux readahead framework. *SIGOPS Oper. Syst. Rev.*, 2008, 42(5): 75-84.

[29] Pai R, Pulavarty B, Cao M. Linux 2.6 performance improvement through readahead optimization. In *Proc. the Linux Symposium*, July 2004, pp.393-403.

[30] Mowry T C, Demke A K, Krieger O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996, pp.3-17.

[31] VanDeBogart S, Frost C, Kohler E. Reducing seek overhead with application-directed prefetching. In *Proc. the USENIX Annual Technical Conference*, June 2009, p.24.

[32] Yang C K, Mitra T, Chiueh T C. A decoupled architecture for application-specific file prefetching. In *Proc. the FREENIX Track: USENIX Annual Technical Conference*, June 2002, pp.157-170.

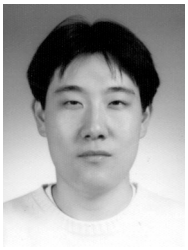
[33] Chang F, Gibson G A. Automatic I/O hint generation through speculative execution. In *Proc. the 3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999, pp.1-14.

[34] Cao P, Felten E W, Karlin A R, Li K. A study of integrated prefetching and caching strategies. In *Proc. the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, May 1995. pp.188-197.

[35] Gill B S, Modha D S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. the USENIX Annual Technical Conference*, April 2005, pp.293-308.

[36] Patterson R H, Gibson G A, Ginting E, Stodolsky D, Zelenka J. Informed prefetching and caching. In *Proc. the 15th ACM Symposium on Operating Systems Principles*, December 1995, pp.79-95.

Yongsoo Joo received the B.S. and M.S. degrees in computer engineering and the Ph.D. degree in electrical and computer engineering from Seoul National University, Seoul, Korea, in 2000, 2002 and 2007, respectively. He is currently a research professor in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. His research interests include embedded systems, nonvolatile memory systems, and I/O performance optimization of storage systems.



Junhee Ryu received the B.S. degree in computer engineering from Korea Aerospace University, in 2003, and the M.S. degree in computer science and engineering from Seoul National University, Korea, in 2005. He is currently working toward the Ph.D. degree in computer science and engineering at the same University. His research interests include file systems, storage systems, and network systems.

Sangsoo Park received the B.S. degree from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1998 and the M.S. and Ph.D. degrees from Seoul National University, Seoul, Korea, in 2000 and 2006, respectively. He is currently an assistant professor in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. His research interests include real-time embedded systems and system software.

Kang G. Shin is the Kevin & Nancy O'Connor Professor of computer science in the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, USA. His current research focuses on computing systems and networks as well as on embedded real-time and cyber-physical systems, all with emphasis on timeliness, security, and dependability. He has supervised the completion of 71 Ph.D.s, and authored/coauthored more than 770 technical articles (more than 270 of these are in archival journals), one textbook and more than 20 patents or invention disclosures, and received numerous best paper awards, including the Best Paper Awards from the 2011 ACM International Conference on Mobile Computing and Networking (MobiCom'11), the 2011 IEEE International Conference on Autonomic Computing, the 2010 and 2000 USENIX Annual Technical Conferences, as well as the 2003 IEEE Communications Society William R. Bennett Prize Paper Award and the 1987 Outstanding IEEE Transactions of Automatic Control Paper Award. He has also received several institutional awards, including the Research Excellence Award in 1989, Outstanding Achievement Award in 1999, Distinguished Faculty Achievement Award in 2001, and Stephen Attwood Award in 2004 from the University of Michigan (the highest honor bestowed to Michigan Engineering faculty); a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and 2006 Ho-Am Prize in Engineering (the highest honor bestowed to Korean-origin engineers). He has chaired several major conferences, including 2009 ACM MobiCom, 2008 IEEE SECON, 2005 ACM/USENIX MobiSys, 2000 IEEE RTAS, and 1987 IEEE RTSS. He is the fellow of both IEEE and ACM, and served on editorial boards, including IEEE TPDS and ACM Transactions on Embedded Systems. He has also served or is serving on numerous government committees, such as the U.S. NSF Cyber-Physical Systems Executive Committee and the Korean Government R&D Strategy Advisory Committee. He has also co-founded a couple of startups.