Spidermine: Low Overhead User-Level Prefetching*

Jiwoong Won, Jemin Ahn
Dept. of Computer Science and Engineering,
Hanyang University, Republic of Korea
{jiwoongwon,ahnjemin}@hanyang.ac.kr

Sangwoon Yun†
Dept. of Computer Science and Engineering,
Hanyang University, Republic of Korea
swyun@hanyang.ac.kr

Jongchan Kim
Dept. of Automobile and IT Convergence,
Kookmin University, Republic of Korea
jongchank@kookmin.ac.kr

Kyungtae Kang‡
Dept. of Computer Science and Engineering,
Hanyang University, Republic of Korea
ktkang@hanyang.ac.kr

ABSTRACT

Spidermine monitors the rate at which read requests are issued by an application, and thus detects bursts of disk reads. It then determines an address at which to insert a breakpoint into the application code or a library before each burst, and logs each breakpoint, together with the data required for the subsequent burst. When the application is subsequently run, Spidermine inserts breakpoints at each logged address by temporarily replacing the instruction. Spidermine is then invoked at each breakpoint, and prefetches the corresponding data blocks into the page cache. This use of breakpoints as triggers for prefetching eliminates the need for monitoring to determine when to prefetch data during program execution. Also, by operating at the user level, Spidermine avoids interference with other applications. Experiments on 11 benchmark applications demonstrated that Spidermine can reduce the time for launch by up to 54.1%, and for run-time data-loading by up to 70.1% on a hard disk drive, 13.3% and 47.0% respectively, on a solid-disk drive.

CCS CONCEPTS

• **Software and its engineering** → **Main memory**; **Software performance**;

KEYWORDS

Spidermine, Prefetching, I/O optimization

ACM Reference Format:
Jiwoong Won, Jemin Ahn, Sangwoon Yun, Jongchan Kim, and Kyungtae Kang. 2023. Spidermine: Low Overhead User-Level Prefetching. In *Proceedings of ACM SAC Conference (SAC'23).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3555776.3577754

*A preliminary version of this work was presented at IEEE SMC'17 and was published in its proceeding [19]
†Major in Bio Artificial Intelligence
‡Corresponding author

SAC'23, March 27 –March 31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577754

1 INTRODUCTION

Application launch times (which we will call "launch times") and the delays that occur when a running application has to load additional data from a disk (which we will call "loading times") reduce the satisfaction experienced by personal computer users [1, 5, 7, 8, 14, 20]. Software developers can ameliorate these delays by the timely prefetching of the disk blocks likely to be needed by an application before they are required [2, 4, 9–11, 13, 16]. For example, operating systems such as Windows and Linux use prefetching internally [13], and there have been many attempts to improve the performance of prefetching. Most prefetchers record the patterns in which disk reads occur during program launch. Then, when the application is subsequently executed, they fetch the corresponding blocks in advance. This approach has been shown to be effective in reducing application launch times, but its benefit in terms of the loading of run-time data has been largely unexamined.

More recently, prefetching techniques have been designed specifically to accelerate the loading of data [3, 6, 12, 15, 18], as well as application launches. Prefetching data after an application has launched is more complicated because there are no obvious indications that data is about to be loaded, and so disk reads must be monitored continuously to determine when to prefetch data. This incurs large CPU and memory overheads and the accuracy with which data is prefetched is often disappointing.

These challenges motivated us to design a lightweight prefetching scheme called Spidermine, which uses breakpoints as triggers. This approach is based on two observations that many personal computing applications read a lot of data from storage in initial launch and later bursts, and that they are likely to request the same data during each run. These observations led us to design Spidermine to recognize bursts of reads.

During its learning and analysis phase, Spidermine runs the application and identifies the bursts of reads by counting the frequency of read operations. This is carried out at the user level without searching for individual block correlations at the kernel level. Consecutive read requests are logged in a circular queue, and a sequence of requests is recognized as a burst if the time to fill the queue is less than or equal to a period `burst_threshold`. This process makes few demands on either the CPU or memory.

Once the bursts of reads have been detected, Spidermine finds locations ahead of each burst in the application code or a library

code at which a breakpoint should be inserted. We use an instruction pointer, which is a register in the processor that stores the address of the next instruction to be executed, to determine the addresses at which to insert breakpoints. When the application is subsequently executed, Spidermine inserts the breakpoints into the application code and libraries as they are loaded from a disk into the memory, and backs up the instructions replaced by each breakpoint. After all of the breakpoints have been inserted, Spidermine starts the application. As the application runs, each breakpoint causes a software interrupt (called a SIGTRAP signal) to be issued. Spidermine responds by prefetching the corresponding data into the page cache so that it becomes available just before a burst of reads. It then restores the original instruction at the breakpoint, and the application resumes execution. No monitoring is required between breakpoints, and the operational overhead is too small to measure. Spidermine is compared with existing prefetchers in Table 1.

We have evaluated Spidermine using a desktop computer with an Intel Core i7-8700 CPU with 32 GB of RAM, a Seagate 3.5-inch 2-TB 7200-RPM HDD, and an Intel 535 Series 250 GB SSD, running Ubuntu 20.04 64-bit Linux with an EXT4 file system. We assessed the effect of varying the effect of the value of `burst_threshold`. Experiments with 11 benchmark applications (Firefox, Eclipse, ONLYOFFICE, LibreOffice, Android Studio, FlightGear v2017, Pillars of Eternity (PoE), Pillars of Eternity II: DeadFire (PoE II), Divinity, The Long Dark, and SOMA) indicate that Spidermine can reduce application launch times by up to 54.1% (Firefox) and run-time loading times by up to 70.1% (PoE) on HDD. It also can reduce application launch times by up to 13.3% (Eclipse) and run-time loading times by up to 47.0% (PoE) on SSD.

The contributions of our work are summarized as follows:

- The design of a prefetcher that expedites both launching and subsequent loading without the need for a large table of correlations, which is suitable for applications such as games that need not only to be launched fast, but also to load significant amounts of data after launching.
- The use of breakpoints as triggers for prefetching to eliminate the need for continuous monitoring to determine when to prefetch data during program execution, keeping Spidermine lightweight.
- The user-level implementation of the merging and sorting of read requests, which leverages the effectiveness of prefetching.
- The use of the user space rather than the kernel space for tracing read operations, analysis, and prefetching so that they do not affect the execution of other applications.

2 SPIDERMINE DESIGN

The effectiveness of prefetching is largely determined by the accuracy with which data are selected, the timelines with which they are fetched, and the overhead incurred in these operations. Existing kernel-level prefetching schemes are mostly focused on reducing application launching times. Or they do attempt to reduce loading times but incur significant CPU and memory overheads in the analysis of the correlations between data blocks. Moreover, obtaining the information from a running application needed to infer block correlations inevitably requires significant changes to the OS kernel, which is intrinsically undesirable and means that the prefetcher must be updated whenever the kernel changes.

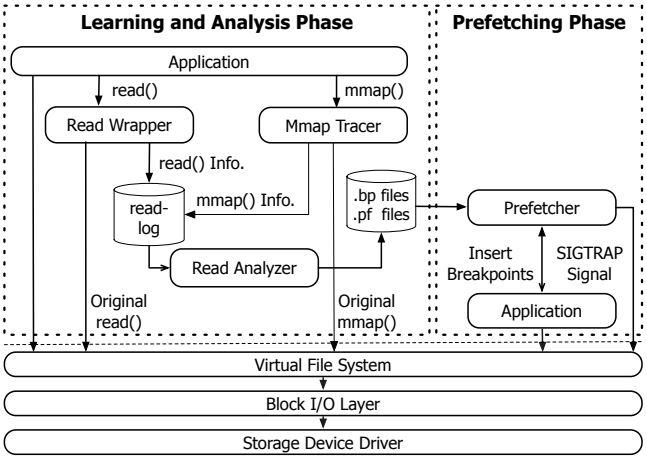


Figure 1: Overall structure of Spidermine.

Spidermine monitors file operations rather than individual block reads, and both monitoring and prefetching are carried out at the user level. This means that it requires minimal modifications to the OS kernel, and these modifications only affect target applications. Moreover, Spidermine inserts breakpoints in the application and library codes that it uses as they are loaded. And each breakpoint triggers prefetching of the blocks which were associated with that breakpoint during the learning and analysis phase. This is much more efficient than continuously monitoring trigger blocks demanded by other prefetchers. We now describe the tracer, analyzer, and prefetcher components that enable Spidermine to operate.

2.1 Read Tracer in the Learning Phase

Fig. 1 depicts the overall structure of Spidermine. The read tracer, which is the logical component of Spidermine, physically consists of an "mmap tracer" and a "read wrapper." The mmap tracer runs the application program instead of a Linux shell, and after that, these two physical components track all of the read-related operations requested by the application (marked as (1) in Fig. 2a) and generate a log for the collected data (marked as (2) in Fig. 2a). The tracer only collects information on requests to read regular files.

One source of reads is the mapping of a file to a region of memory. This is performed by the `mmap()` system call, after which the file can be accessed just like an array. This is more efficient than `read()` because only the parts of the file that the program actually accesses are loaded. The common use of memory mapping is for the loading of a dynamic library. Since it incurs a large number of disk reads, the run-time loading of dynamic library is our focus. The mmap tracer uses the `ptrace()` system call to trace read requests that result from `mmap()` called with PROT_EXEC or PROT_READ flag, which maps a dynamic library to the virtual memory of a process. And it examines the address space to which the dynamic library is mapped. It is also noted that the memory mapping segment of each library itself may contain many disk-read operations.

The reading of data from a disk is more straightforwardly performed by `read()` or `fread()` in the C library. Reads of this sort can be traced using a dynamic library we have implemented, called

Table 1: Comparison of Spidermine with existing prefetchers

Scheme	Scenario	I/O optimization	Memory overhead	CPU overhead	Performance	Implementation
GSoC Prefetch [13]	Launch	File-level sorting	Low	Mid	Mid	Kernel level
Preload [4]	Launch	Not applicable	Low	Low	Low	User level
C-Miner [12]	Launch & Loading	LBN sorting	Mid	Mid	Low	Kernel level
DiskSeen [3]	Launch & Loading	LBN sorting	High	Mid	Mid	Kernel level
ClusterFetch [15]	Launch & Loading	LBN sorting	Low	Low	High	Kernel level
Application-directed Prefetching [18]	Launch & Loading	LBN sorting	Low	Low	Depending on Programmer's Expertise	User level
Spidermine	Launch & Loading	LBN sorting	Low	Low	High	User level

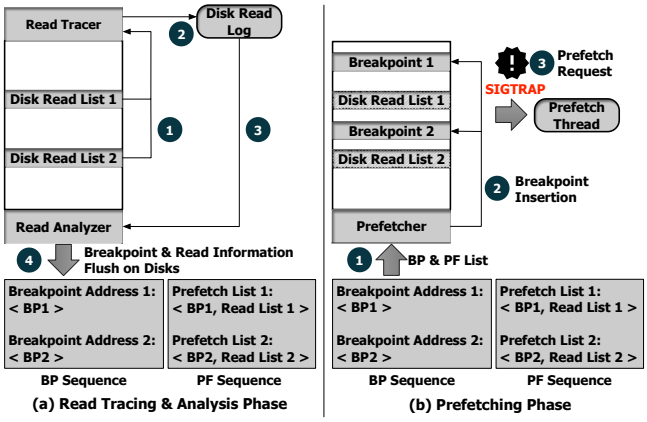


Figure 2: Operation of Spidermine.

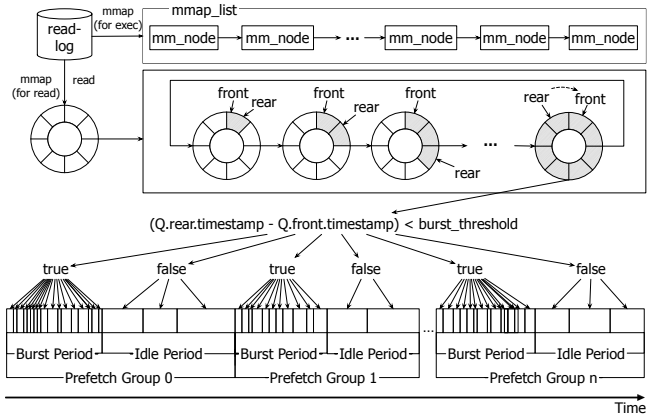


Figure 3: Identifying and grouping bursts and idle periods.

the read wrapper, which provides wrapper functions for these functions. We use this approach to avoid the overhead of repeatedly stopping and starting the target process by calls to `ptrace()`. Later, when the application calls `read()` or `fread()`, the read wrapper intercepts these calls and traces the corresponding reads. This technique requires the function being called to have an entry in the procedure linkage table (`.plt` section) of the C library `libc`. After recording the read request information, the read wrapper calls the original function from `libc.so` that the application can continue executing normally.

However, reads made by `mmap()` cannot be intercepted in this way because `mmap()` is not listed in the linkage table; therefore, the slower `ptrace()` approach must be used.

2.2 Read Analyzer

Spidermine is designed to recognize bursts of reads rather than trying to correlate individual blocks. The read analyzer is the component of Spidermine (marked as (3) in Fig. 2a) that uses the log information obtained during tracing to identify the bursts of reads by determining the frequency of read operations, which it logs in a circular queue. This approach has low CPU and memory overheads.

If the difference between the times when the first and last entries were logged into the circular queue is shorter than the predefined threshold `burst_threshold`, then the read analyzer decides that these entries correspond to a burst of disk reads over a period equal to the time difference. Bursts that follow one another clearly are combined, and the remaining times are classified as idle periods. They are also combined if they are only separated by a few reads. Fig. 3 shows how bursts of reads and idle periods are identified and how they are linked into prefetch groups. Note that a burst period and the idle period that follows it become a "prefetch group." We choose this design instead of linking each idle period with the subsequent burst, as a breakpoint in a particular group triggers the prefetching of the data requested in the subsequent prefetch group.

After classification, the analyzer examines the return addresses of the functions called in each idle period and selects the first address at which it is suitable (the details of how to determine this will be described in Section 3.2.2) to insert a breakpoint. The breakpoint should be inserted as near as possible to the start of an idle period, which allows as much time as possible for loading the blocks required during the following burst of reads. Finally, the analyzer stores sequence information (SI) on the disk (marked as (4) in Fig. 2a). This includes 1) the offset of each breakpoint from the starting memory address of the application code or a library code, which we call the breakpoint (BP) sequence, and 2) a tuple (file name, start offset, length of the read) at the breakpoint that characterizes the data to be prefetched, which we call the prefetching (PF) sequence.

Although the location of an application in virtual memory and the locations of dynamic libraries loads may change each time it is executed, the location of each breakpoint can be easily adjusted using the offset. An operational overview of the read tracer and analyzer is summarized in Fig. 2a.

2.3 Prefetcher

After the tracer and analyzer performed the learning and analysis phase during the first execution of a program, the prefetcher carries out prefetching on the basis of the schedule learned by the tracer and analyzer from the second time of execution. Similar to the mmap tracer, the prefetcher runs the application program on behalf

of the Linux shell. The prefetcher loads the application and SI into memory (marked as (1) in Fig. 2b) and then inserts breakpoints at the addresses designated by the BP sequence (marked as (2) in Fig. 2b). The existing operation code (opcode) at each such address is replaced by the opcode INT 3 (i.e., 0xCC), which is designed to set breakpoints for debugging. The original opcode is then stored in the memory space of the prefetcher, which then executes the application and waits to encounter breakpoints in the form of INT 3 opcodes, as discussed above.

Each breakpoint raises a SIGTRAP signal, which causes Spidermine to create a thread to fetch the data corresponding to the entry in the PF sequence that is associated with the breakpoint (marked as number 3 in Fig. 2b). After each prefetching thread has been started, the original opcode is restored, and the application resumes execution. The restoration of the original opcode is particularly important when the breakpoint is inserted into the library that is shared with other applications, to avoid their execution of an instruction with the wrong opcode. An operational overview of the proposed prefetcher design is summarized in Fig. 2b.

3 IMPLEMENTATION

As described in Section 2, Spidermine consists of the read tracer, read analyzer, and prefetcher. We will now describe how these components were implemented in Linux 5.9.

3.1 Read Tracer: Tracing of Disk Reads

We have slightly modified `ptrace()` and the kernel so that the kernel responds only to the `mmap()` system call and ignores other system calls. In addition, we have implemented the read wrapper to hook calls to two read-related functions, `read()` and `fread()`, which trace disk reads.

3.1.1 Mmap-tracer: Tracing reads caused by run-time library loading. The mmap tracer, which is a component of the read tracer, executes the target application as a child process and monitors all of the threads it creates. The mmap tracer uses `ptrace()` to trace the disk reads incurred when loading the executable binary image of the application and run-time dynamic libraries that it calls. Essentially, `ptrace()` is a system call that allows a tracer process to observe and control the execution of a tracee process and to examine or change the memory and register information used by the tracee. The prototype of `ptrace()` is as follows:

```
long ptrace(enum __ptrace_request request, pid_t pid,
    void *addr, void *data);
```

The value of `request` determines an action to be performed, and `pid` identifies the tracee by its thread. A tracee first needs to be attached to the mmap tracer, which can be initiated by the mmap tracer invoking `fork()` to create the tracee as a child process. The mmap tracer then calls `ptrace(PTRACE_SEIZE, pid, 0, PTRACE_O_MMAPTRACE)`, where `PTRACE_SEIZE` is an action which makes a process specified in `pid` attached to the mmap tracer, and `PTRACE_O_MMAPTRACE` is a newly defined option to deactivate tracing of all system calls except for `mmap()`.

After `ptrace()`, any kernel-generated signal delivered to the tracee (with the exception of SIGKILL) will result in the stoppage of the tracee. In the kernel mode, the tracee invokes a `ptrace_notify()`,

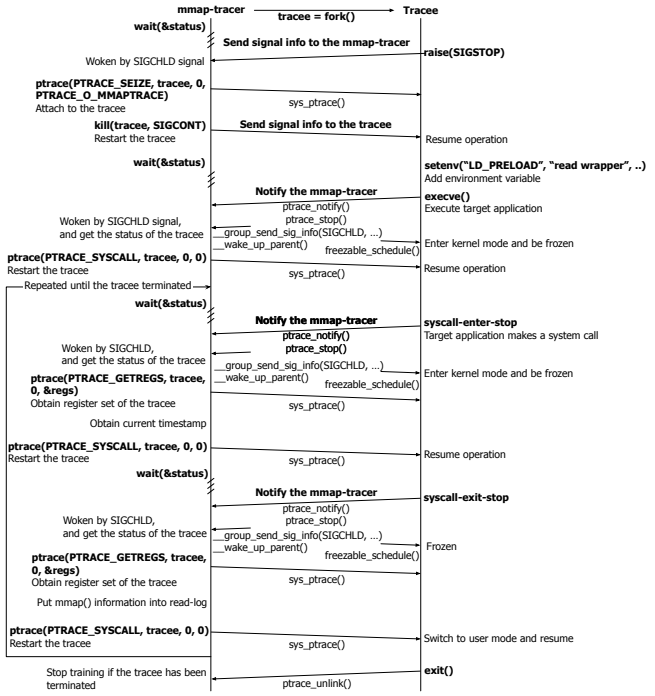


Figure 4: Interactions between the mmap tracer and the tracee.

which subsequently calls `ptrace_stop()`; this calls `__group_send_sig_info(SIGCHLD,..)`, followed by functions `__wake_up_parent()` and `freezable_schedule()`.

These calls stop the tracee and send a SIGCHLD signal to the mmap tracer, which receives the signal from a call to `wait()` where a status value showing the cause of the stoppage can be also retrieved. While the tracee is stopped, the mmap tracer can make various requests to `ptrace()` to inspect and modify the tracee. The mmap tracer then calls `ptrace(PTRACE_SYSCALL,tracee,0,0)` to tell the kernel to stop the tracee whenever it enters or leaves a system call, and waits until the tracee stops.

After resuming its operation by `PTRACE_SYSCALL`, the tracee enters the **syscall-enter-stop** state upon entry to a system call. The mmap tracer then collects the information about that system call, and restarts the tracee, again by calling `ptrace(PTRACE_SYSCALL,...)`. Upon leaving the system call, the tracee enters the **syscall-exit-stop** state. This repeated pausing of the tracee at the entry and exit of every system call may seem to incur a large amount of overhead. However, since Spidermine does not need to stop the tracee for system calls other than `mmap()`, its impact to the system is minimal.

When the mmap tracer invokes `PTRACE_SYSCALL` to restart the tracee, the `TIF_SYSCALL_TRACE` flag is set on the data structure `thread_info` of the tracee. If this flag is set, the kernel stops the tracee and activates the tracer through `ptrace_notify()`, when it handles system calls from the tracee. The newly defined option `PTRACE_O_MMAPTRACE` ensures that the kernel stops the tracee and activates the tracer for `mmap()` system calls only. This small modification to the kernel reduces the tracing overhead without affecting the behavior of any other applications.

When the tracee enters the **syscall-enter-stop** or **syscall-exit-stop** state, the mmap tracer is woken and immediately calls `ptrace()` again to obtain the data in the registers used by the tracee:

```
ptrace(PTRACE_GETREGS, tracee, 0, &regs);
```

where `&regs` is the address of the `user_regs_struct` data structure that receives the contents of these registers. The data in `user_regs_struct` allow the tracer to record the `mmap()` information. After all `mmap()` system calls have been processed, the tracer stores the `mmap()` information in a "read-log" file for later use by the analyzer. The operation of the mmap tracer is described in Fig. 4.

3.1.2 Read wrapper: Tracing reads caused by read-related functions. Preloading is a feature of the Linux dynamic linker on most systems. It allows the user to specify a shared library to be loaded before all of the other shared libraries using the environment variable `LD_PRELOAD`. The mmap tracer adds `read_wrapper.so` to the variable when it starts up.

The shared object `read_wrapper.so` contains customized `read()` and `fread()` codes that trace disk reads requested by the original `read()` and `fread()`. When an application calls `read()` or `fread()`, the linker checks all of the preloaded libraries first and then finds libraries that have been linked to an application. We have therefore designed these wrapper functions to hook the original ones, so as to collect information about disk reads.

These wrapper functions eventually call `dlsym()`, which accepts the "handle" to a dynamic library returned by `dlopen()` and a null-terminated symbol name, and returns the address of that symbol in memory. If `dlsym()` is given a special pseudo-handle, `RTLD_NEXT`, it finds the next occurrence of a function in the search order after the current library that actually corresponds to the original `read()` or `fread()` function in `libc.so`.

Finally, the wrapper functions call the original function in `libc.so` by following the address returned by `dlsym()` and store the size (`ret`) of the original `read()` in the "read-log" file with the time stamp and function arguments that have already been collected.

3.1.3 Read wrapper: Tracing breakpoint candidates. The read wrapper also traces addresses returned by four C library functions `strlen()`, `memcpy()`, `memmove()`, and `strcpy()`. We measured the number of calls for each function in `libc.so` that were invoked by our benchmark applications using `ltrace`, an open-source library tracing program, and empirically found out that those four functions are most commonly called during application execution (see Supplemental Material for `ltrace` results). These return addresses are considered as candidates for breakpoints, some of which are later selected for actual breakpoints as discussed later. To trace these addresses, we constructed wrapper functions of these four functions. Each wrapper function hooks calls to one of these four functions, calls the original function in `libc.so` by following the address returned by `dlsym()`, and records their return addresses into a "candidate-log" file along with the time stamp upon returning.

3.2 Read Analyzer

After the application terminates, the read tracer passes control to the read analyzer, which detects burst periods on the basis of the information collected in the learning phase. The way in which the analyzer detects bursty and continuous disk reads is described in Algorithm 1, which uses a circular queue to log all disk reads.

Algorithm 1 Detection of read bursts

```
1: function analyze_log(read-, candidate-logs)
2:     while is_empty(read-log) ≠ false do
3:         read_entry ← get_one_entry(read-log)
4:         if is_mmap(read_entry) then
5:             if is_prot_exec(read_entry) then
6:                 append_mmap_list(read_entry)
7:             end if
8:         end if
9:         enqueue(read_entry, Q)
10:        if is_full(Q) then
11:            insert_queue_into_Pgroup(Q, Pgroup)
12:            if is_burst(Q) = true then
13:                previous_period ← BURST
14:            else
15:                if is_triggered(Pgroup) = false then
16:                    if set_trigger(Q, candidate-log, mmap_list, Pgroup) = true
   then
17:                        Pgroup.next ← new_Pgroup()
18:                        Pgroup ← Pgroup.next
19:                    end if
20:                end if
21:                previous_period ← IDLE
22:            end if
23:            reset_queue(Q)
24:        end if
25:        continue
26:    end while
27: end function
28: function is_burst(Q)
29:    if (Q.rear.TS - Q.front.TS) < burst_threshold then
30:        return true
31:    end if
32:    return false
33: end function
```

3.2.1 Detection of read bursts. The Spidermine analyzer reads the logs written by the read tracer during the learning phase to determine when and what to prefetch and to schedule the data to be prefetched. The read analyzer reads the logs for `mmap()` and the read-related functions from the read-log file (using `get_one_entry()`), in the ascending order of the time stamp, and inserts them into a circular queue. Additionally, the analyzer maintains `mmap_list` in memory to store the `mmap()` information collected so far, as shown in Fig. 3, using `append_mmap_list()`. When the circular queue is full, the analyzer calls `insert_queue_into_Pgroup()`. This is a function that inserts the information on disk read requests into `Pgroup`, which is a data structure that tracks all of the meta-information about a prefetch group, such as the location of the breakpoint (`Pgroup.breakpoint`) within the idle period and the read requests (`Pgroup.reads`) created in that group during the idle and burst periods. These information are maintained with a linked list (see Fig. 5).

Before all the meta-information is copied to `Pgroup`, adjacent read requests are merged into a smaller number of larger requests, followed by sorting with their physical offsets to maximize the read performance of HDDs (see Section 3.4). After inserting all the meta-information into `Pgroup`, the analyzer obtains the difference between the time stamps at the rear (`Q.rear.TS`) and front (`Q.front.TS`) of the queue to determine the time it took for the queue to fill up, which is then compared with the predefined threshold `burst_threshold` to detect a burst period.

Algorithm 2 Selection of target addresses for breakpoints

```
1: function append_mmap_list(mmap_entry, mmap_list)
2:     mmap_list.tail ← mmap_entry
3:     mmap_list.tail.next ← new_mmap_node()
4:     mmap_list.tail ← mmap_list.tail.next
5: end function
6: function is_triggerred(Pgroup)
7:     if Pgroup.bp_offset = -1 and Pgroup.md = -1 then
8:         return false
9:     end if
10:    return true
11: end function
12: function set_trigger(Q, candidate-log, mmap_list, Pgroup)
13:    mmap_node = mmap_list.head
14:    while is_empty(candidate-log) ≠ true do
15:        bp_entry ← get_one_entry_from_candidates()
16:        while mmap_node ≠ NULL do
17:            mmap_path ← mmap_node.path
18:            mmap_start_addr ← mmap_node.start_addr
19:            mmap_end_addr ← mmap_node.end_addr
20:            candidate_addr ← bp_entry.return_addr
21:            if (mmap_start_addr ≤ candidate_addr) && (candidate_addr ≤
    mmap_end_addr) then
22:                Pgroup.md ← hash(mmap_path)
23:                Pgroup.bp_offset ← candidate_addr - mmap_start_addr
24:                return true
25:            end if
26:            mmap_node ← mmap_node.next
27:        end while
28:    end while
29:    return false
30: end function
```

If the difference is less than the threshold, the interval between these two time stamps is classified as the burst period containing a sufficiently large number of burst reads (otherwise, it corresponds to an idle period). Then, the read analyzer correlates that burst period with a prefetch group, initializes the rear index of the queue (`reset_queue()`), and fills read request information from the front of the queue sequentially. If the queue is full again, the analyzer appends the information on the queued read requests to the existing `Pgroup`, merges and sorts them, and determines whether it is a burst period. If burst periods are continuously detected, they are aggregated into a single burst period, and `Pgroup` is updated accordingly. In a similar way, if successive idle periods are detected following the burst period, they are also combined, and `Pgroup` is updated accordingly. When a new burst period is detected after that idle period, a new prefetch group is created using `new_Pgroup()`, and the information on read requests is now associated with that prefetch group.

Repeating these steps results in the creation of successive prefetch groups, each of which represents a combination of burst and idle periods, as shown in Fig. 5. We again manage these prefetch groups with the linked list for processing efficiency.

3.2.2 Determining the target addresses of breakpoints. After the disk reads have been classified into burst and idle periods, the read analyzer looks for an address at which a breakpoint can be inserted in an idle period. This is done by examining the return addresses of `strlen()`, `memcpy()`, `memcmp()`, and `malloc()` in the candidate-log file.

We consider these return addresses in order of increasing distance from the start of each idle period, as a breakpoint near the start of an idle period gives the prefetcher a better chance to have

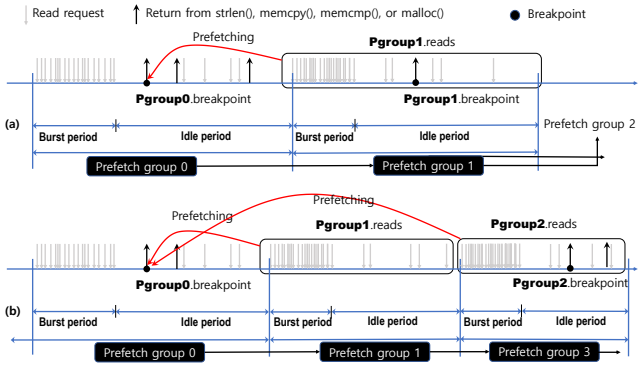


Figure 5: List of prefetch groups for the management of breakpoints and read requests.

all of the data ready in time. However, to become a breakpoint, a return address must be unique so that it can be unambiguously linked with a sequence of disk reads. Thus, calls made during different idle periods that all return to the same address cannot be used. In particular, when a function is called indirectly through other library functions, the return addresses of that function could be the same in many cases regardless of locations from which that particular function was called.

Algorithm 2 describes how breakpoint target addresses are determined on the basis of these guidelines. When an idle period is detected, the read analyzer searches the candidate-log file in the order of time stamp to check the return addresses of four C library functions, and selects the first found in between `Q.front.TS` and `Q.rear.TS` of the idle period as the breakpoint. Then, the analyzer passes this return address along with the `mmap_list` as an argument to the `set_trigger()` function. The `set_trigger()` function compares the mapping space of the applications or libraries stored in the `mmap_list` with the passed return address to determine the mapped file and the breakpoint offset (BPO)[1] from the base mapping address of that file. This information about the breakpoint is saved into `Pgroup`.

A prefetch group i may be linked back to a breakpoint in a group that comes earlier than $(i-1)$) if group $(i-1)$ contains no suitable place for a breakpoint (i.e., if no return address is found in the idle period), as shown in Fig. 5b. Fig. 6 depicts how a breakpoint is selected among return addresses, where the return address 0x7f2ede6da845 is selected as the breakpoint. This address falls within a library that was mapped from 0x7f2ede5c7000 to 0x7f2ede9a3b08. The BPO, which is 0x113835 in this case, is determined from the base mapping address of the library.

The analyzer repeats the process of filling the queue, checking for bursts, and creating prefetch groups for all the logs in the read-log file. After that, two SI files with suffixes **.bp** and **.pf** are generated. The **.bp** file contains a pair <MD, BPO>for each breakpoint, where MD is the message digest, which is the hashed value of a library path. The **.pf** file contains entries of MD, BPO, path, file offset, and length. Here, path, file offset, and length are for the data to be prefetched at each breakpoint in the **.bp** file.

[1]It should be noted that we need to use an offset because `mmap()` may allocate a different memory space to a dynamic library each time the application runs.

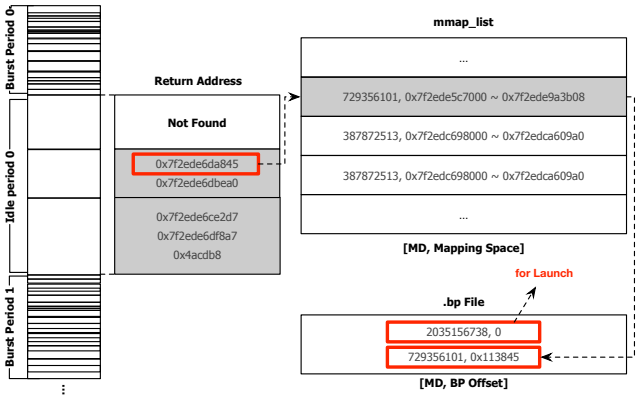


Figure 6: Selection of breakpoint addresses.

3.3 Prefetching Data

The prefetcher loads the application code into memory and inserts all the breakpoints in the **.bp** file, including those from run-time dynamic libraries. The prefetcher then overwrites the opcodes of instructions in the breakpoints found in the **.bp** file with 0xCC, and stores the opcodes that have been replaced. Next, it loads the data that is required right at the launch, using `posix_fadvise()`, with the information from the **.pf** file. After prefetching, the opcode of the original instruction is restored, and the application is started by the prefetcher.

As the application starts running, the prefetcher calls `ptrace()` and waits for `mmap()` to be called to load dynamic library files. The prefetcher and the application respectively become the tracer and tracee following the same protocol of Fig. 4. Once the application and the libraries are loaded into memory, the prefetcher can insert relevant breakpoints found in the **.bp** file into the text segment by adding their offsets to the base mapping address of both the application and the libraries. Again the original opcodes are stored and replaced with 0xCC. Each of these breakpoints, whether in the application itself or a library, triggers the creation of a thread that runs concurrently to prefetch the data associated with that breakpoint, without interrupting application execution. The original opcode should be restored as well.

3.4 Read Optimization: Merging and Sorting

Spidermine is designed to improve the efficiency of prefetching process by minimizing the number of read requests and reducing disk search time. This can be done by merging small read requests into a single large one and reordering those requests by the LBN [17]. Unlike kernel-level prefetchers that can directly receive LBN information, a user-level prefetcher Spidermine cannot directly identify the LBNs of read requests. To get the LBN-related information, the read tracer instead utilizes `fiemap ioctl()` that returns file extent mapping information, where we can also extract the physical offset from the start of the extent. This physical offset is equivalent to the LBN for prefetching purposes. The read analyzer subsequently uses the physical offset for merging and sorting before generating the **.pf** file. For SSDs, Spidermine only performs merging, preserving the original sequence of read logs, due to sorting has little impact on throughput.

4 PERFORMANCE EVALUATION

We have evaluated Spidermine on a desktop computer with Intel Core i7-8700 CPU, 32GB of RAM, a Seagate 3.5inch, 2 TB, 7200RPM HDD, and an Intel 535 Series 250 GB SSD, running Ubuntu 20.04 64-bit Linux, with the EXT4 file-system. For the detection of read bursts, the number of entries available in the circular queue was 32, and the value of the threshold `burst_threshold` was set to 3.0 s, which was also varied between 0.1 and 7 s to investigate its effect on launch and loading times.

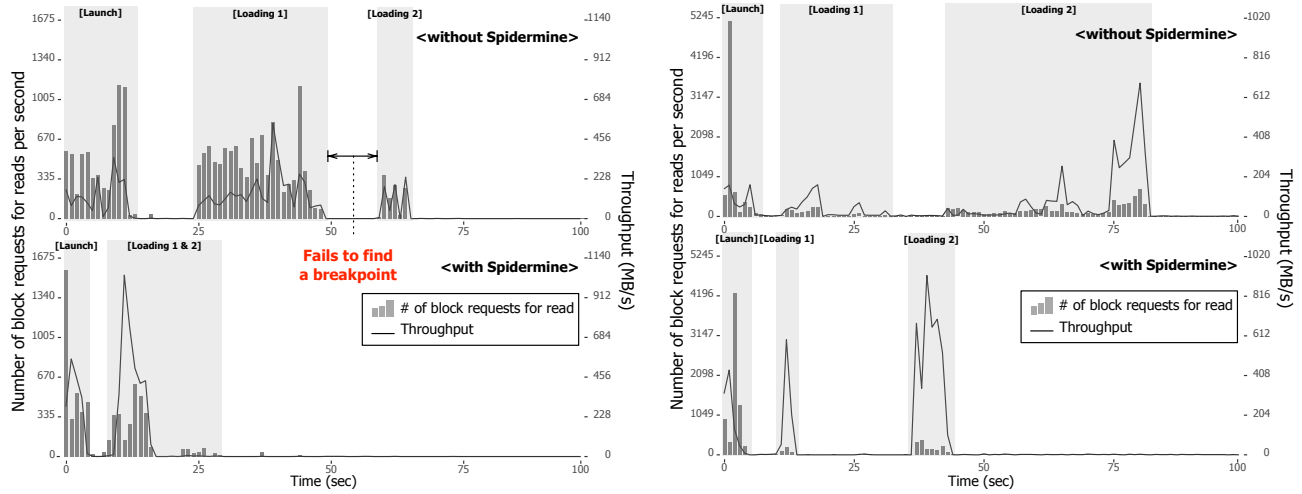
4.1 Launch and Loading Times

Fig. 7 shows the pattern of the number of requested blocks and the throughput monitored in the block I/O layer of the Linux kernel during the execution of two game applications on HDD: Pillars of Eternity (PoE) and SOMA. In a cold start, no application-related data is available in the page cache when an application launches. Conversely, in a warm start, all data already resides in the page cache. In the cold start scenario of Fig. 7a, there are bursts of reads with idle periods between them. The first burst is associated with application launching, during which Spidermine reduces the rate of requests for blocks, while the throughput increases. This increase can lead to the merger of adjacent small reads into a smaller number of larger read requests, which are sorted by their physical offsets from the beginning of the disk.

Initially, Spidermine increases launching time because two SI files must be loaded from the disk, breakpoints have to be inserted into the application code, and data associated with the launch must be prefetched into the memory, all before starting the application. During the launching of Firefox and Eclipse, the startup overhead due to this were 3.7 and 3.1 s respectively. However, by caching all the launch-related data into the page cache, subsequent interrupts and disk reads can be avoided. Launching is also expedited by the merging and sorting of reads. Overall, Spidermine reduces the launch time of Firefox by 54.1%, and that of Eclipse by 42.9%, despite the startup overhead (see Table 2). In the case of SSD, application execution resumes immediately after requesting a prefetching thread, and therefore there is almost no overhead.

The bursts of reads after the launch are associated with the loading of additional data. There is an idle period before each burst which allows a lot of prefetching, as shown in Fig. 7a. In the run of PoE, Spidermine reduces the number of read requests that occur between 25 and 48 s (loading 1), and also between 58 and 65 s (loading 2), thanks to the data availability in the page cache. A similar pattern can be observed in the run of SOMA, which is shown in Fig. 7b.

We compared the application launch and loading times of 11 applications in three scenarios: cold start with and without Spidermine, and warm start. The warm start is equivalent to an ideal prefetching process, which loads all and only the correct blocks in zero time. Table 2 shows the effect of Spidermine on application launch and loading times. We see that Spidermine reduces cold start launch times by between 12.1% (Divinity) and 54.1% (Firefox), and run-time loading times by between 13.8% (Divinity) and 70.1% (PoE) on HDD. On SSD, it reduces cold start launch times by between 0.7% (The Long Dark) and 13.3% (Eclipse), and run-time loading times by between 4.5% (Divinity) and 47.0% (PoE). On average, Spidermine



(a) Pillars of Eternity (b) SOMA

Figure 7: Number of blocks read per second and throughput, observed at the block I/O layer of the Linux kernel in a cold start, with and without Spidermine, for two sample applications. In (a), the read analyzer fails to find a suitable place for a breakpoint in the idle period between loading 1 and loading 2, which is similar to the case shown in Fig. 5b.

Table 2: Experimental results on application launch and loading times (averaged over 10 iterations).

Application	W (s)	C on HDD (s)	C_T on HDD (s)	R on HDD (%)	O on HDD (s)	C on SSD (s)	C_T on SSD (s)	R on SSD (%)	O on SSD (s)
Firefox (launch)	1.1	15.7	7.2	54.1	3.7	1.7	1.6	5.9	0.01
Eclipse (launch)	4.7	23.1	13.2	42.9	3.1	6.0	5.2	13.3	0.02
Only Office (launch)	1.8	18.0	13.8	23.3	1.8	2.6	2.3	11.5	0.01
LOImpress (launch)	1.6	16.9	11.2	33.7	2.5	2.4	2.1	12.5	0.01
Android Studio (launch)	8.1	42.2	34.8	17.5	1.9	9.5	8.5	10.5	0.07
FlightGear (launch)	23.9	37.9	29.7	21.6	4.3	27.1	26.4	2.6	0.02
PoE I (launch)	8.8	13.6	11.3	16.9	3.9	10.1	9.6	5.0	0.02
PoE II (launch)	21.2	39.6	32.5	17.9	2.3	24.3	23.6	2.9	0.02
Divinity (launch)	8.1	17.3	15.2	12.1	0.8	9.4	9.1	3.2	0.01
The Long Dark (launch)	11.1	26.8	22.6	15.7	1.2	13.6	13.5	0.7	0.02
SOMA (launch)	4.1	9.8	7.2	26.5	2.0	5.6	5.4	3.6	0.01
Average reduction on HDD				25.7		Average reduction on SSD		6.5	
The Long Dark (loading 1)	5.8	9.2	6.2	32.6		6.8	5.9	13.2	
Divinity (loading 1)	8.2	11.6	10.0	13.8		8.9	8.5	4.5	
Divinity (loading 2)	8.3	13.9	11.1	20.1		9.5	8.8	7.4	
PoE I (loading 1)	5.6	27.8	8.3	70.1		11.7	6.2	47.0	
PoE I (loading 2)	3.5	9.4	4.5	52.1	NA	6.2	4.3	30.6	NA
PoE II (loading 1)	7.4	53.5	16.9	68.4		11.2	9.5	15.2	
PoE II (loading 2)	6.1	25.8	11.6	55.0		9.4	7.8	17.0	
SOMA (loading 1)	6.4	20.2	16.0	20.8		10.4	8.9	14.4	
SOMA (loading 2)	9.8	43.3	13.9	67.9		16.2	13.7	15.4	
Average reduction on HDD				44.6		Average reduction on SSD		18.3	

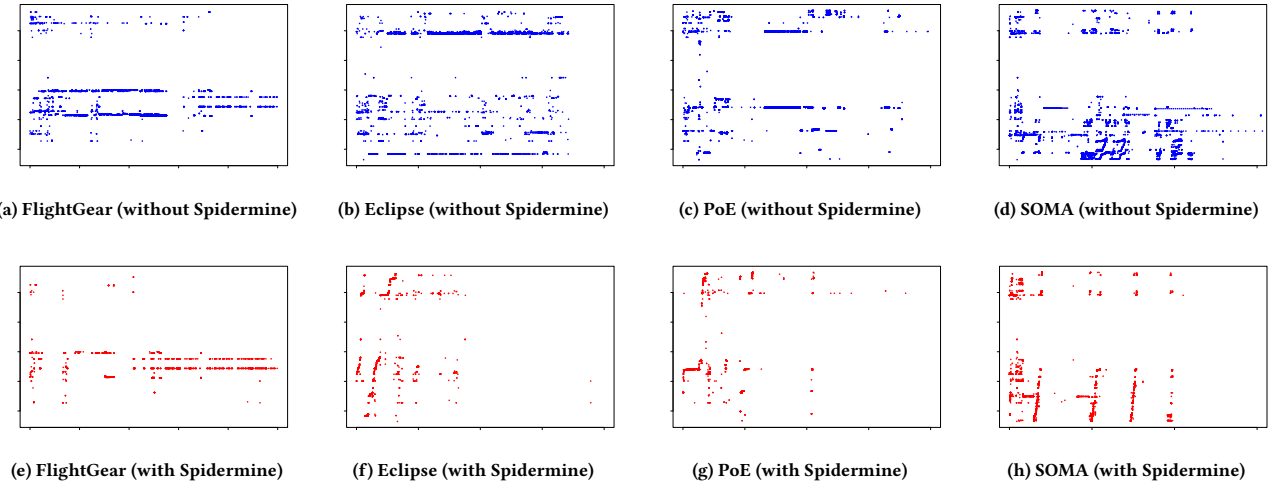
C: Cold start without Spidermine, $\mathbf{C}_T$: Cold start with Spidermine
W: Warm start, **R**: Reduction, **O**: Startup overhead of Spidermine

reduces launch times by 25.7% and loading times by 44.6% on HDD, and launch times by 6.5% and loading times by 18.3% on SSD, which are comparable to ClusterFetch [15], a state-of-the-art kernel-level prefetcher.

4.2 Prefetching Effectiveness

In application launching, it is important for the prefetcher to efficiently read data that will be requested by the application. In particular, prefetching is always performed just before the application launch and block request patterns hardly vary across different launches. Fig. 8a and Fig. 8b show access patterns for data blocks during the launch of FlightGear and Eclipse, both of which indicate that many blocks are requested and access patterns are almost random. Spidermine reduces the number of block requests and induces the HDD to read sequentially, by optimizing reads through merging and sorting. Fig. 8e and Fig. 8f show the results of these optimizations. Here, each point represents the first block number of a request, and a reduction in the number of points implies the effectiveness of merging. We can also confirm the effectiveness of sorting by observing increasing point patterns in Fig. 8e and Fig. 8f. In Fig. 8f, little data is requested after the launch. However, in Fig. 8e, data is continuously requested even after the launch, which is due to the characteristic of FlightGear having a different background each time it runs.

In application loading, the required data should be prefetched in a timely manner, and prefetching should not affect the execution of the application. Spidermine achieves accurate prefetching by classifying the application period into burst and idle periods and inserting breakpoints in the idle periods. In addition, Spidermine creates a separate prefetch thread at the time of prefetching, so as



(a) FlightGear (without Spidermine) (b) Eclipse (without Spidermine) (c) PoE (without Spidermine) (d) SOMA (without Spidermine)

(e) FlightGear (with Spidermine) (f) Eclipse (with Spidermine) (g) PoE (with Spidermine) (h) SOMA (with Spidermine)

Figure 8: Block read patterns observed at the block I/O layer of the Linux kernel in a cold start with and without Spidermine, for four sample applications. Time is displayed on the horizontal axis and LBN (929523712~1936914432) on the vertical axis.

not to disturb the execution of the application. Fig. 8c and Fig. 8d show block access patterns when PoE and SOMA are launched without Spidermine and run for a certain period of time. These applications require lots of data during run time and we can observe that Spidermine successfully performs merging and sorting for the loading of data in Fig. 8g and Fig. 8h.

4.3 Effect of Burst Detection on Performance

We varied the value of `burst_threshold` to determine its impact on launch and loading times. A smaller threshold value would result in fewer burst periods and more idle periods; whereas a larger threshold value more burst periods and fewer idle periods. When `burst_threshold` is too small, many idle periods incur many breakpoints, which can be inserted into non-ideal locations. Too many attempts at prefetching in wrong places lead to increase in launch and loading times. In the extreme case, there can be no burst period at all, which means all the data should be prefetched in the launch. This can severely increase the launch time, as shown in Fig. 9, the threshold at 0.1 s.

On the contrary, when `burst_threshold` is too large, it is expected that prefetching is triggered with just a few breakpoints, which means that a large amount of data should be prefetched at each breakpoint. This can also impact prefetching performance as shown in Fig. 9, where the launch time surges after the threshold at 3 s. It appears that in Fig. 9, 3 s is the optimal choice for `burst_threshold`, as it can reduce both launch and loading times for PoE. This analysis demonstrates how important it is to carefully choose an application-specific threshold value. On SSDs, the performance is much better than HDDs, so the change of loading times according to the threshold was not large.

4.4 Overhead of the Mmap Tracer

Table 3 shows the number of `ptrace_stop()`, loading times, and the overhead resulted from the mmap tracer in the learning phase. As discussed in Section 3.1.1, `PTRACE_O_MMAPTRACE` deactivates the tracing of all system calls except for `mmap()`. And we are interested in how much overhead can be reduced by using this option, compared with stopping the tracee for every system call. In Table 3, `PTRACE_O_MMAPTRACE` reduces the number of stops of applications during run time by an average of 99.6%. This means that the mmap tracer can dramatically reduce the overhead compared with a traditional method of tracing all system calls, which can be identified by the loading times in Table 3, where the system call trace shows the overhead between 2.0–33.8 s, while the mmap trace between 0.3–22.1 s on HDD. On SSDs, the systcall trace has an overhead of 1.5 to 11.3 s while the mmap trace has an overhead of 0.5 to 6.7 s.

5 CONCLUSIONS

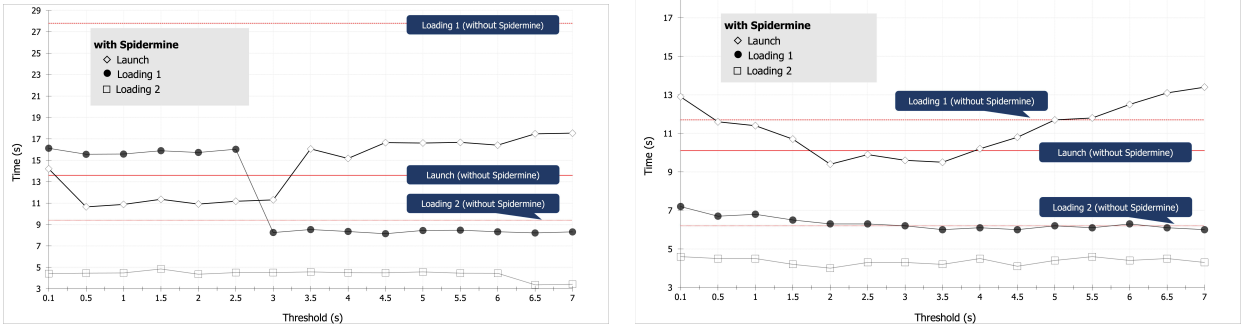
Spidermine is a user-level lightweight prefetcher that reduces application launch and data loading times. Spidermine identifies bursts of reads, and uses this information to plan prefetching. It inserts breakpoints into idle periods between bursts and these breakpoints trigger the prefetching of corresponding disk blocks into the page cache. Spidermine does not continuously monitor applications, keeping CPU overhead low. Furthermore, since Spidermine runs at the user level, it has no effect on other applications. Experiments on 11 popular benchmark applications showed that launch times were reduced by between 12.1% and 54.1%, and loading times by between 13.8% and 70.1% on HDD. On SSD, Spidermine can reduce the time for launch by up to 13.3%, and for run-time data-loading by up to 47.0 on SSD.

ACKNOWLEDGMENTS

This work was partly supported by the IITP grant (No. RS-2022-00155885, AI Convergence Innovation Human Resources Development (Hanyang University ERICA)) and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2022R1F1A1074505).

Table 3: Experimental results for tracing overhead in learning phase (averaged over 10 iterations).

Application	Number of ptrace_stop()			Loading Times (HDD / SSD)			Overhead (HDD / SSD)		
	syscall trace	mmap() trace	Reduction (%)	Cold start (s)	syscall trace (s)	mmap() trace (s)	syscall trace (s)	mmap() trace (s)	Reduction (%)
Firefox (launch)	1410105	4469	99.7	15.7 / 1.7	23.1 / 3.5	16.8 / 2.3	7.4 / 1.8	1.1 / 0.6	85.1 / 66.7
Eclipse (launch)	1953041	8134	99.6	23.1 / 6.0	35.5 / 9.9	25.5 / 7.6	12.4 / 3.9	2.4 / 1.6	80.6 / 59.0
ONLYOFFICE (launch)	965079	7526	99.2	18.0 / 2.6	21.8 / 4.5	18.7 / 3.3	3.8 / 1.9	0.7 / 0.7	81.6 / 63.2
LibreOffice (launch)	161742	4534	97.2	16.9 / 2.4	19.6 / 4.4	17.2 / 3.1	2.7 / 2.0	0.3 / 0.7	88.9 / 65.0
Android Studio (launch)	1337971	5304	99.6	42.2 / 9.5	49.1 / 11.3	45.9 / 10.4	6.9 / 1.8	3.7 / 0.9	46.4 / 50.0
FlightGear (launch)	11765399	2112	100.0	37.9 / 27.1	71.7 / 32.6	41.4 / 29.7	33.8 / 5.5	3.5 / 2.6	89.6 / 52.7
PoE (launch)	468127	3812	99.2	13.6 / 10.1	17.7 / 14.5	14.3 / 11.7	4.1 / 4.4	0.7 / 1.6	82.9 / 63.6
PoE II (launch)	990225	2388	99.8	39.6 / 24.3	43.3 / 30.1	40.3 / 26.9	3.7 / 5.8	0.7 / 2.6	81.1 / 55.2
Divinity (launch)	1454234	20706	98.6	17.3 / 9.4	31.0 / 13.3	27.2 / 10.7	13.7 / 3.9	9.9 / 1.3	27.7 / 66.7
The Long Dark (launch)	3598943	2150	99.9	26.8 / 13.6	38.7 / 17.7	28.2 / 15.1	11.9 / 4.1	1.4 / 1.5	88.2 / 63.4
SOMA (launch)	1272126	1739	99.9	9.8 / 5.6	31.6 / 16.9	27.3 / 12.3	21.8 / 11.3	17.5 / 6.7	19.7 / 40.7
The Long Dark (loading 1)	1341209	68	100.0	9.2 / 6.8	11.4 / 13.2	9.7 / 9.9	2.2 / 6.4	0.5 / 3.1	77.3 / 51.6
Divinity (loading 1)	2731323	23702	99.1	11.6 / 8.9	38.7 / 16.1	33.7 / 13.8	27.1 / 7.2	22.1 / 4.9	18.5 / 31.9
Divinity (loading 2)	1570339	15144	99.0	13.9 / 9.5	22.4 / 14.3	18.6 / 11.5	8.5 / 4.8	4.7 / 2.0	44.7 / 58.3
PoE I (loading 1)	833806	1629	99.8	27.8 / 11.7	31.3 / 13.2	29.8 / 12.2	3.5 / 1.5	2.0 / 0.5	42.9 / 66.7
PoE I (loading 2)	416152	993	99.8	9.4 / 6.2	11.4 / 8.1	10.0 / 7.1	2.0 / 1.9	0.6 / 0.9	70.0 / 52.6
PoE II (loading 1)	1837807	184	100.0	53.5 / 11.2	69.0 / 19.3	57.8 / 13.9	15.5 / 8.1	4.3 / 2.7	72.3 / 66.7
PoE II (loading 2)	1586749	108	100.0	25.8 / 9.4	35.3 / 15.2	28.5 / 11.1	9.5 / 5.8	2.7 / 1.7	71.6 / 70.7
SOMA (loading 1)	2877912	3814	99.9	20.2 / 10.4	33.2 / 18.7	29.9 / 15.4	13.0 / 8.3	9.7 / 5.0	25.4 / 39.8
SOMA (loading 2)	3958141	4299	99.9	43.3 / 16.2	50.1 / 21.8	47.6 / 19.1	6.8 / 5.6	4.3 / 2.9	36.8 / 48.2
Average reduction			99.6	Average reduction					61.6 / 56.6



(a) HDD **(b) SSD**

Figure 9: Launch and loading times for varying values of `burst_threshold` during PoE run.

REFERENCES

[1] A. Bovenzi, J. Alonso, H. Yamada, S. Russo, and K. S. Trivedi. 2013. Towards fast OS rejuvenation: an experimental evaluation of fast OS reboot techniques. In *in Proc. 24th IEEE International Symposium on Software Reliability Engineering*. 61–70.

[2] A. D. Brown, T. C. Mowry, and O. Krieger. 2001. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems* 19, 2 (May 2001), 111–170.

[3] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. 2004. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *in Proc. USENIX 2004 Annual Technical Conference*. 173–186.

[4] B. Esfahbod. 2006. *Preload—An Adaptive Prefetching Daemon*. Master's thesis. University of Toronto, Canada.

[5] B. Hubert. 2005. On faster application startup times: cache stuffing, seek profiling, adaptive preloading. In *in Proc. Ottawa Linux Symposium*. 245–248.

[6] S. Jiang, X. Ding, Y. Xu, and K. Davis. 2013. A prefetching scheme exploiting both data layout and access history on disk. *ACM Transactions on Storage* 9, 3, Article 10 (August 2013).

[7] Y. Joo, J. Ryu, S. Park, and K. G. Shin. 2011. FAST: quick application launch on solid-state drives. In *in Proc. 9th USENIX Conference on File and Storage Technologies (FAST)*. 259–272.

[8] H. Kim, N. Agrawal, and C. Ungureanu. 2012. Revisiting storage for smartphones. In *in Proc. 10th USENIX Conference on File and Storage Technologies (FAST)*. 209–222.

[9] T. M. Kroeger and D. D. E. Long. 1996. Predicting file system actions from prior events. In *in Proc. USENIX 1996 Annual Technical Conference*. 319–328.

[10] T. M. Kroeger and D. D. E. Long. 2001. Design and implementation of a predictive file prefetching algorithm. In *in Proc. USENIX 2001 Annual Technical Conference*. 105–118.

[11] H. Lei and D. Duchamp. 1997. An analytical approach to file prefetching. In *in Proc. USENIX 1997 Annual Technical Conference*. 275–288.

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

[13] K. Lichota. 2007. Prefetch: Linux solution for prefetching necessary data during application and system startup. https://code.google.com/p/prefetch/

[14] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *in Proc. 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 275–284.

[15] J. Ryu, D. Lee, K. G. Shin, and K. Kang. 2018. ClusterFetch: A lightweight prefetcher for intensive disk reads. *IEEE Trans. Comput.* 67, 2 (September 2018), 284–290. https://doi.org/10.1109/TC.2017.2748939

[16] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. 2002. Track-aligned extents: Matching access patterns to disk drive characteristics. In *in Proc. 1st USENIX Conference on File and Storage Technologies*. 259–274.

[17] V. Tarasov, G Sim, A Povzner, and E Zadok. 2012. Efficient I/O scheduling with accurately estimated disk drive latencies. In *in Proc. 8th annual workshop on Operating Systems Platforms for Embedded Real-Time*. 36–45.

[18] Steve VanDeBogart, Christopher Frost, and Eddie Kohler. 2009. Reducing seek overhead with application-directed prefetching. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. 299–312.

[19] J. Won, O. Kwon, J. Ryu, J. Hur, I. Lee, and K. Kang. 2017. A Breakpoint-based Prefetcher for BothLaunch and Run-time. In *in Proc. IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2766–2771.

[20] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. 2012. Fast app launching for mobile devices using predictive user context. In *in Proc. 10th International Conference on Mobile Systems, Applications, and Services*. 113–126.