FSVP Project Report

Sukanto Ghosh(07305801)
Ganesh Wagle(07305805)
Suhas Kajbaje(07305906)

Computer Science and Engineering Department
Indian Institute of Technology
Bombay,India.

1 Introduction to the code

Readahead is a widely deployed technique to bridge the huge gap between the characteristics of disk drives and the inefficient usage by applications. At one end, disk drives are good at large sequential accesses and bad at seeks. At the other, applications tend to do a lot of tiny reads. To make the two ends meet, modern kernels and disk drives do readahead: to bring in the data before it is needed and try to do so in big chunks.Readahead typically involves actively detecting the access pattern of all read streams and maintaining information about them.

2 Description

We are verifying *readahead.c* file from linux kernel: *(http://lxr.linux.no/linux+v2.6.22.14/mm/readahead.c)*.

The heuristics in the function *readahead.c* can be summarized in **four** aspects:

2.1 Sequential Detection

If the first read at the start of a file, or a read that continues from where the previous one ends, assume a sequential access. Otherwise it is taken as a random read.

2.2 Readahead Size

There are three phases in a typical readahead sequence:

1. *initial*: When there exists no current_window or ahead_window, the size of initial readahead is mainly inferred from the size of current read request. Normally readahead_size will be 4 or 2 times read_size.

2. *ramp-up*: When there is a previous readahead, the size is doubled or x4.

3. *full-up*: When reaching max_readahead.

2.3 Readahead Pipelining

To maximally overlap application processing time and disk IO time, it maintains two readahead windows: current_window is where the application expected to be working on; ahead_window is where asynchronous IO happens. ahead_window will be openedrenewed in advance, whenever it sees a sequential request that-

1. is oversize
2. has only current_window
3. crossed into ahead_window

2.4 Cache hit/miss

A *readahead cache hit* happens when a page to be readahead is found to be cached already.When the threshold VM_MAX_CACHE_HIT(=256) is reached, readahead will be turned off to avoid unnecessary lookups of the page-cache.

A *readahead cache miss* happens when a page that was brought in by readahead is found to be lost on time of read.

3 Property verification

If the read mode is sequential, and in such case if the pages to be readahead are found to be in cache itself(i.e. cache hit) in large numbers, the readahead mode should be disabled. This property is not obvious on inspection of code because the decision to turn off RA mode depends unpo the number of pages found in cache itself.
During the study of code we came to know that there was a bug in this module about "accidental Read Ahead off". The span of the bug was not limited to the file we delt with. Thus we zeroed on this property which could be thought of as a subproperty of the bug.

4 Problems faced while analyzing the code

The only viable source of infomation about the specification of linux kernel is the comments present in the source code itself. We made an exaustive search through kernel source files. Thus we had to put efforts in getting a code, that

made sense from verification point of view. However, after we zeroed upon the code, we found an elegent research paper on the module "Page Cache Readahead".

5 Problems encountered while verifying the code using BLAST

1. The code uses few Struct elements and the instances are accessed using pointers to these struct elements. Using the -talias option in BLAST the pointers could be handled.

2. However, We could not put asserertion involving conjunction between two or more predicates. e.g $assert(ra \rightarrow size == 0 \&\& ra \rightarrow flags == 0)$. It is important to note that in the code, the struct instance is passed by reference, to other functions and the its elements' values are getting modified in functions two levels down the main caller function. To make things easier we wrote a dummy program to make assertion of these type. But BLAST was able to assert on only one predicate at the time. We used different predicate generators, such as -craig, -clp; but that couldn't help. This does not remain a problem if the we are not dealing with the pass by reference progamming style.

3. We couldn't provide predicates through .pred file which involve condition on struct element which are accessed via pointers. e.g. $(ra \rightarrow size! = 0)$ couldnt be provided to BLAST, and we didnt find any syntax, so as to how provide it. Again; we created a dummy program to check this.

4. There were few bitwise operations in the code, those couldnt be handled by tool. But those operations made no impact on the properties we were verifying.

6 Changes made in the original source code

1. The file we looked into is mm/readahead.c. In the beginning we commented out the code that was not relevent to our analysis. However as we moved on we restored the original source file to atmost extent.

2. We added few constructs (struct and macro) those were declared in other file and were indeed used in the readahead.c such as:

Structs : address_space, file_ra_state, file
Macros : min, roundup_pow_of_two

7 Conclusion

1. BLAST is much more powerful than what we explored for this exercise, however its available documentation underestimates its significance.

2. Finding a code was indeed a standalone exercise; however we got to learn few nudgets of linux kernel.