New Readahead in CLIO (bug 20294)

Jay

2010.1.13

1 Introduction

There're many drawbacks in the current implementation of readahead which is ported from b1_8. It doesn't work well under some scenarios as follows:

- No stripe boundary detecting, this causes small RPCs to be sent
- it detects read pattern at ->readpage, this is not good because it's easy to get wrong idea of read pattern
- too agreesive if the application consumes the data slowly, this causes readahead pages evicted in vain under memory pressure
- readahead is dependent on application, but it does not allow applications to have their own readahead module, even doesn't have an interface to disable lustre readahead

In this document, a new readahead scheme is proposed under CLIO, which is hopeful to address the above issues.

2 Requirements

- optimizing RPC size
- adapative readahead window based on the consuming speed of applications
- more precise read pattern detecting
- plugin module support so that customers' can have their own readahead
- WNC support
- Improve read performance

3 Definitions

- readahead window

Readahead window is purely determined by read pattern, it can only be updated in ll_file_read and ll_nopage. But there is an exception, if miss_in_window event occurs, the window len will set to half size in ll_readpage. It's defined in ll_readahead_state{}.

- readahead size

Because of many restrictions, we can't issue as many pages as readahead window described. So readahead size is used to describe the size we issued once. readhead size is defined in ra_io_arg{}.

- lazy readahead

If applications consume data slower than the speed of reading data from OSTs, we don't need to bring in too much data in memory. In this case, lazy readahead can be enabled. That is to say, though the read pattern is quite good, we just read ahead the data by the speed of consuming data by applications.

- lazy window size

If lazy readahead is enabled, we know that the read(from OSTs) speed is faster than the consume speed. But it might as well has a reasonable size of page-in-cache region, so that in case the application happens to read more data, we are still able to provide enough data. This region is described as lazy_window_size, it's defined in ll_readahead_state{}.

4 Functional specification

I'm going to cover what we will do in this section.

4.1 Read pattern detect

In order to get precise read pattern, we should detect it near the syscalls. So we're going to detect the readahead window in ll_file_read and ll_nopage. Also, we'll store multiple seek histories so that it's easier to know the pattern of seeks.

Because read pattern is tightly coupled with applications, it's impossible to know every kinds of read patterns(only applications know). We'll only have sequential read pattern supported internal(probably have reverse sequential read supported). For other kinds of read pattern, we just provide some interfaces so that customizing application-specific readahead is easy.

4.2 layered readahead size detect

In lustre, because of dlm lock coverage, ptlrpc congestion and optimizing RPC size, it may not issue as many pages as readahead window to OSTs. Instead it has to determine thee readahead size by asking each layer.

In the new readahead implementation, we're going to pass on readahead window to lov and osc to get a suitable read size. For example, in LOV layer, it detects if the read ahead window is aligned to stripe boundary, and in OSC layer, it detects the dlm lock coverage and if the ptlrpc is in congestion.

There'll be sync and async readahead request. If the page is missed in readahead window, apparently we have to read that page as soon as possible - we call this kind of requests 'sync readahead request'. For those requests used to streamline the read, we call them 'async readahead request'. We do not issue an async request if we can't compose a full size RPC.

4.3 Lazy readahead

The essence of readahead is that when the application needs to read the next page, the page has already been in memory, so it makes no sense to bring in as many pages as poosibe to memory, if applications consume data slowly. Obviously we shouldn't read ahead too aggressive in this case, because the readahead pages would be evicted in vain if the memory is under pressure. We need to invent an algorithm to detect this case, and then adjust the readahead size accordingly.

We'll detect this case by measuring the speed of application consuming the data, says consume_speed, and the speed of reading the pages from OSTs, says read_speed. If read_speed is larger than consume_speed, and if we have already had enough pages after read position, we do not need to read far too much.

4.4 Plugin readahead module

Detecting actual read patterns is proven to be difficult, only applications themselves know the exact read pattern. It's good to have an interface to allow customers to implement their own readahead algorithm. At lease, we have to provide an interface to disable per file readahead so that they can do it in user space.

As I mentioned, we're going to have only one system internal read pattern detector - sequential read. For stride IO pattern, we're going to implement it as an external module.

4.5 WNC support

This may not be easy. We probably have to revise ll_file_read to not call system specific routines to read data. For example, in Linux, we depend on generic_file_read and then readpage to read data, we have to revise it so as to issue the read request in ll_file_read directly. This is open to discuss.

4.6 Lock ahead

If the read pattern is really good, it makes sense to lock file priorly. But this has many aspects to be considered, for example, what if the objects on OSTs were quite busy, we can't disturb the application on other clients; what if the lock ahead missed, we then need a punlishment mechanism. This is open to discuss.

5 Use cases

We'll generate the following read patterns, and investigate the readahead behavior.

5.1 Small sequential read

5.2 Large sequential read

5.3 Application consumes data slowly

5.4 Application consumes data quickly

5.5 OST is quite busy

5.6 random read

6 Logic specification

6.1 Prototypes

6.1.1 ll_readahead_state

```
struct ll_readahead_state {
    /** The data to remember the last continuous read, by pages */
    pgoff_t       ras_prev_off;
    unsigned long ras_prev_count;
    /** historical seeks, last 4 times. Negative # means seek backward */
    int64_t       ras_seeks[4];
    /** curpos of ras_seeks */
    int           ras_seek_pos;
    /** number of read requests after the last seek. */
    unsigned long ras_consecutive_requests;
    /** Readahead window start and length */
    unsigned long ras_window_start, ras_window_len;
    /** Where the last read-ahead ended, so it's the start point of next read */
    unsigned long ras_next_readahead;
    /** lazy readahead is enabled if the speed of consuming data is slower then speed c
     * and has this # of pages in cache */
    unsigned long ras_lazy_window;
    /** lazy readahead marker point */
    unsigned long ras_lazy_marker;
    unsigned long ras_lazy_mark_page;
}
```

6.1.2 ra_io_arg

```
/**  cl_io_ra_arg is a container which includes the readpage and readahead
 * parameters. This data structure is passed onto <vvp, lov, osc> layer
 * to determine the read size.  */
struct ra_io_arg {
    /** start page index of readahead window */
    unsigned long ria_start;
    /** end page index of readahead window */
    unsigned long ria_end;
    /** # of page slots reserved in global readahead cache control. */
    long ria_reserved;
    /** set if the reading page was evicted because of memory pressure. */
    int ria_miss_in_window;
    /** a point to weigh how busy is the ptlrpc. */
    int ria_congestion;
}
```

6.1.3 ll_readahead_update

This function records the read history data, and analyze the read pattern. If plugin module is supported, it scans the read pattern module list and calls each detectors. It also runs Lazy readahead detecting algorithm.

```
void ll_readahead_update(struct file *file, pgoff_t pgidx, ulong pgcnt, int vfs_or_mmap
{
    if (readahead is disabled for this file)
        return;
    if (pgidx is sequential to <ras_prev_off, ras_prev_count>) {
        ras_prev_count <- ras_prev_count + pgcnt;
    } else {
        ras_seeks[ras_seek_pos] <- pgidx - ras_pref_off;
        ras_seek_pos <- next pos;
    }
    /* Run lazy readahead algorithm */
    ...
    /* Adjust readahead window */
    if (sequential) {
        increase the readahead window by RAS_INCREASE_STEP;
    } else {
            reset the readahead window, set readahead window to <pgidx, pgcnt>
        }
    }
```

6.1.4 ll_readahead

This function is used to create pages by ra_io_arg, then add them into read queue. Before creating pages, it checks if the page is covered by a dlm lock via ->cpo_is_under_lock().

6.1.5 ll_io_readpage

ll_io_readpage is called by ll_readpage to read a page into memory. It calls cl_io_read_page which then calls each layer's ->cio_readpage method to determine the suitable readahead size. Finally ll_io_readpage then creates pages by the readahead size, and submit the IO.

```
    /* @page is the page being read. */
    int ll_io_readpage(..., struct cl_io *io, struct cl_page *page, struct file *file)
    {
        initializes ra_io_arg @ra;
        add the @page into read queue;
        call cl_io_readpage(..., io, page, ra);
        call ll_readahead to create pages by @ra and add them into read queue;
        revoke unused budget; /* reserved in vvp_io_readpage */
        submit read queue;
    }
```

6.1.6 cl_io_readpage

```
    int cl_io_readpage(struct lu_env *env, struct cl_io *io, struct cl_page *page, strucr r
```

6.1.7 vvp_io_readpage

vvp_io_readpage first checks if the reading page has defer-uptodate marked, if this is the case, it means the page is already in cache; then it determines the readahead size by ll_readahead_state, also, it reserves the budget from ra in sbi.

```
    int vvp_io_readpage(..., struct page *page, struct ra_io_arg *ra)
    {
        get ccc_page from @page;
        if (!ccc_page->cp_defer_uptodate) {
            ra->ria_miss_in_window = 1;
        }
    /* Get the readahead size */
    ra->ria_start = ras_next_readahead;
    ra->ria_end   = ras_window_start + ras_window_len - 1;
    reserve budget;
}
```

6.1.8 lov_io_readpage

If the request will be an async request(ria_miss_in_window is false), lov_io_readpage makes sure that the readahead size is aligned to stripe boundary. lov_io_readpage will be complex.

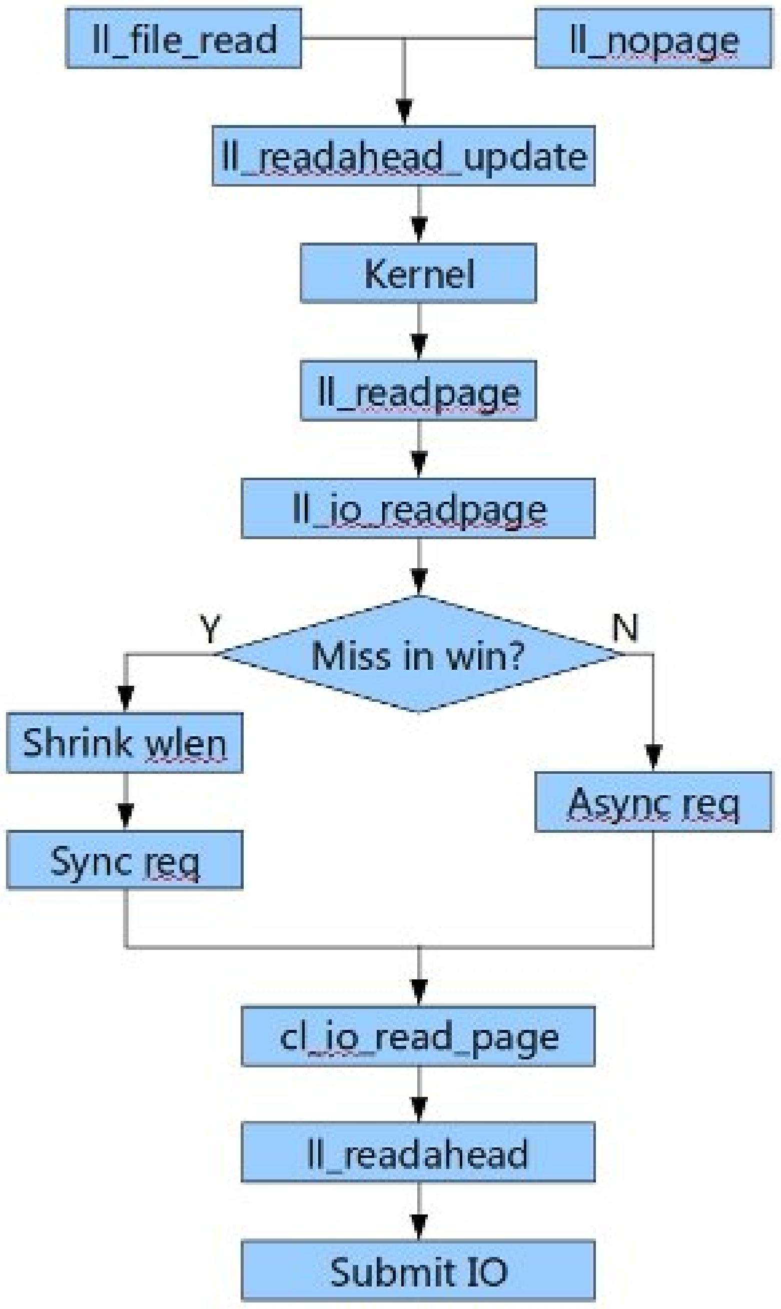
```
int lov_io_readpage(..., struct cl_page *page, struct ra_io_arg *ra)
{
    for (each stripe covered by @ra) {
        compose stripe ra, says stripe_ra;
        call cl_io_readpage against the stripe;
        if (stripe_ra->ria_end is not on the stripe boundary) {
            adjust @ra accordingly;
            break;
        }
    }
    if (async readahead request && @ra size is less then a full RPC size) {
        /* Do not send RPC */
        ra->ria_end = ra->ria_start - 1;
    }
}
```

6.1.9 osc_io_readpage

osc_io_readpage checks the dlm lock coverage of <ria_start, ria_end>, and shrink this region accordingly. Also, it checks if the ptlrpc is in congestion, it adjusts the size by policy as well.

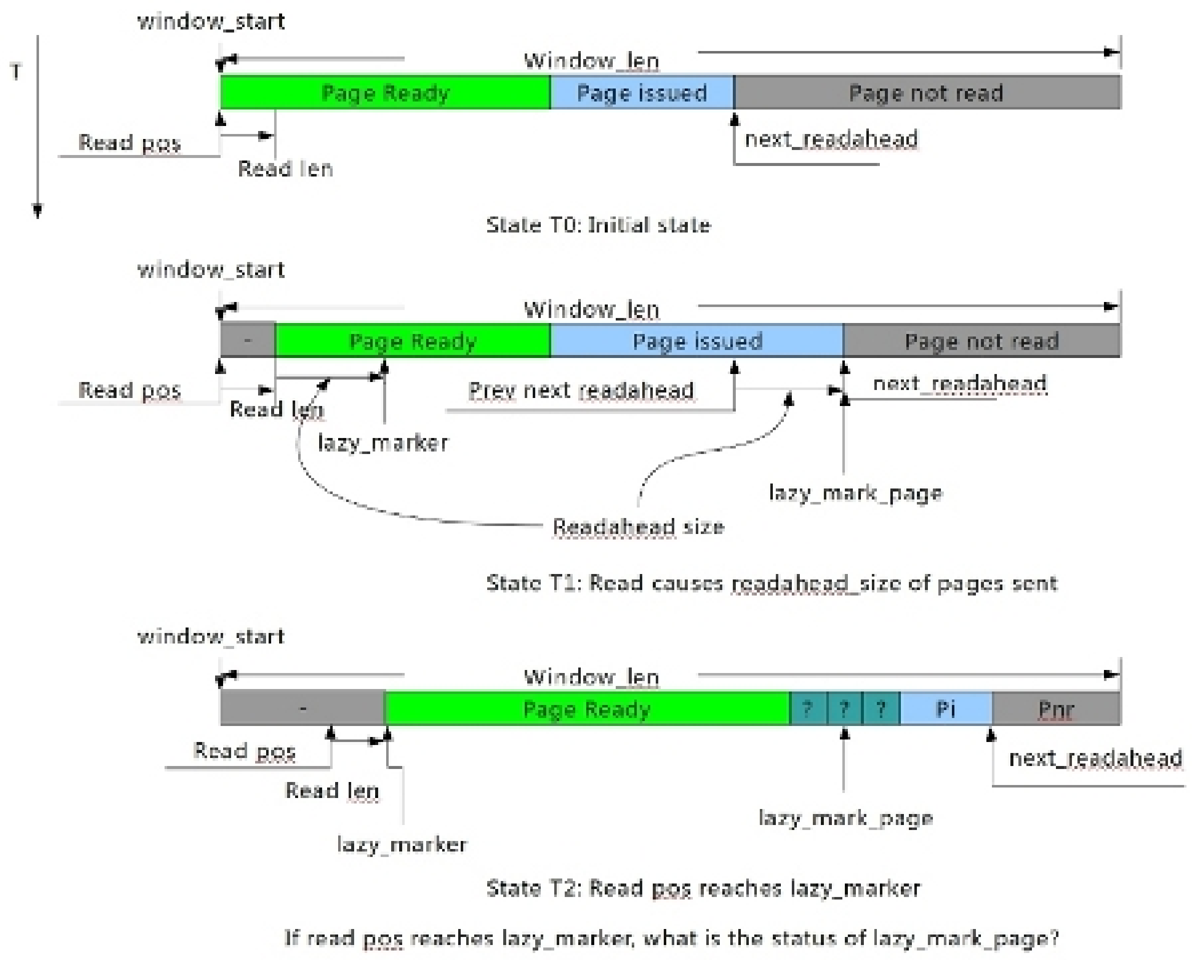
```
int osc_io_readpage(..., struct cl_page *page, struct ra_io_arg *ra)
{
    /* adjust @ra by the coverage of dlm lock; */
    if (@ra is not fully covered by dlm lock) {
        adjust @ra accordingly;
    }
    /* set ra->ria_congestion by ptlrpc queue status */
    if (ptlrpc->in_flights < 8) {
        ra->ria_ptlrpc_congestion = Green;
    } else if (ptlrpc->in_flights < 16) {
        ra->ria_ptlrpc_congestion = Yellow;
        ra->ria_end = ra->ria_start + ...;
    } else {
        ra->ria_ptlrpc_congestion = Red;
        ra->ria_end = ria_start - 1;
    }
}
```

6.2 Flow chart



6.3 An algorithm of lazy readahead

Here is a way to estimate the speed of consuming data by application as follows:



As the picture shows:

1. Have a lazy_marker and lazy_mark_page;
2. when readahead happens, assuming it issued readahead_size pages, then lazy_mark_page is used to remember the last page just issued, and lazy_marker is used to remember the distance of (read_pos + read_len + readahead_size);
3. when the read_pos reaches lazy_marker, we can then check if the lazy_mark_page is ready. If it's ready, we can say that the read speed if a bit faster then consuming speed, and vice versa;

4. Set lazy_window to be the distance between next_readahead and read_pos(we can weigh the previous lazy_window to avoid thrashing). If the lazy_window is reasonable big, and read_speed is larger then consume_speed, we can start lazy_readahead;

5. For lazy readahead, bascially we issue one RPC unless the application has consumed one-full-RPC size of data.

6.4 Plugin module support

T.B.D

7 State management

8 Alternatives

8.1 Readahead thread per file

To address the problem that we issue the readahead RPC in the reading process, this is considered to time consuming. We have an idea to spawn a per file readahead thread for each process, and this thread can be used to issue the readahead RPC async.

However, it has some problems about this scheme:

- context switch is expensive
- if the client is busy, we can't issue the RPCs in time. This causes readahead is actually not usable.
- faireness - it'd not be reasonable to have two threads for serving read request. This is unfair to other processes in system.

9 Focus for inspections