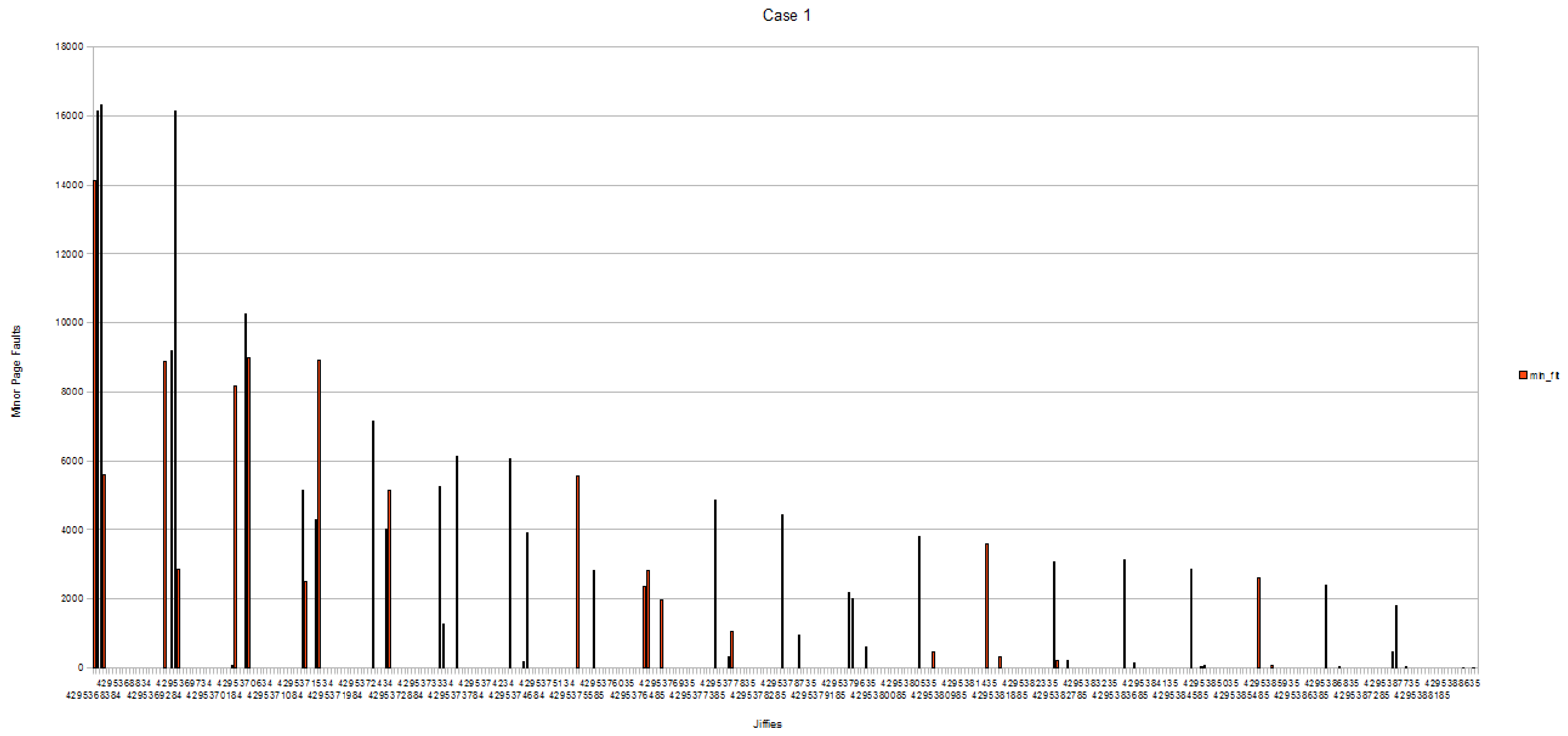
CS 423 – Fall 2011 Group 6: Burrough, Busjaeger, Decker MP3

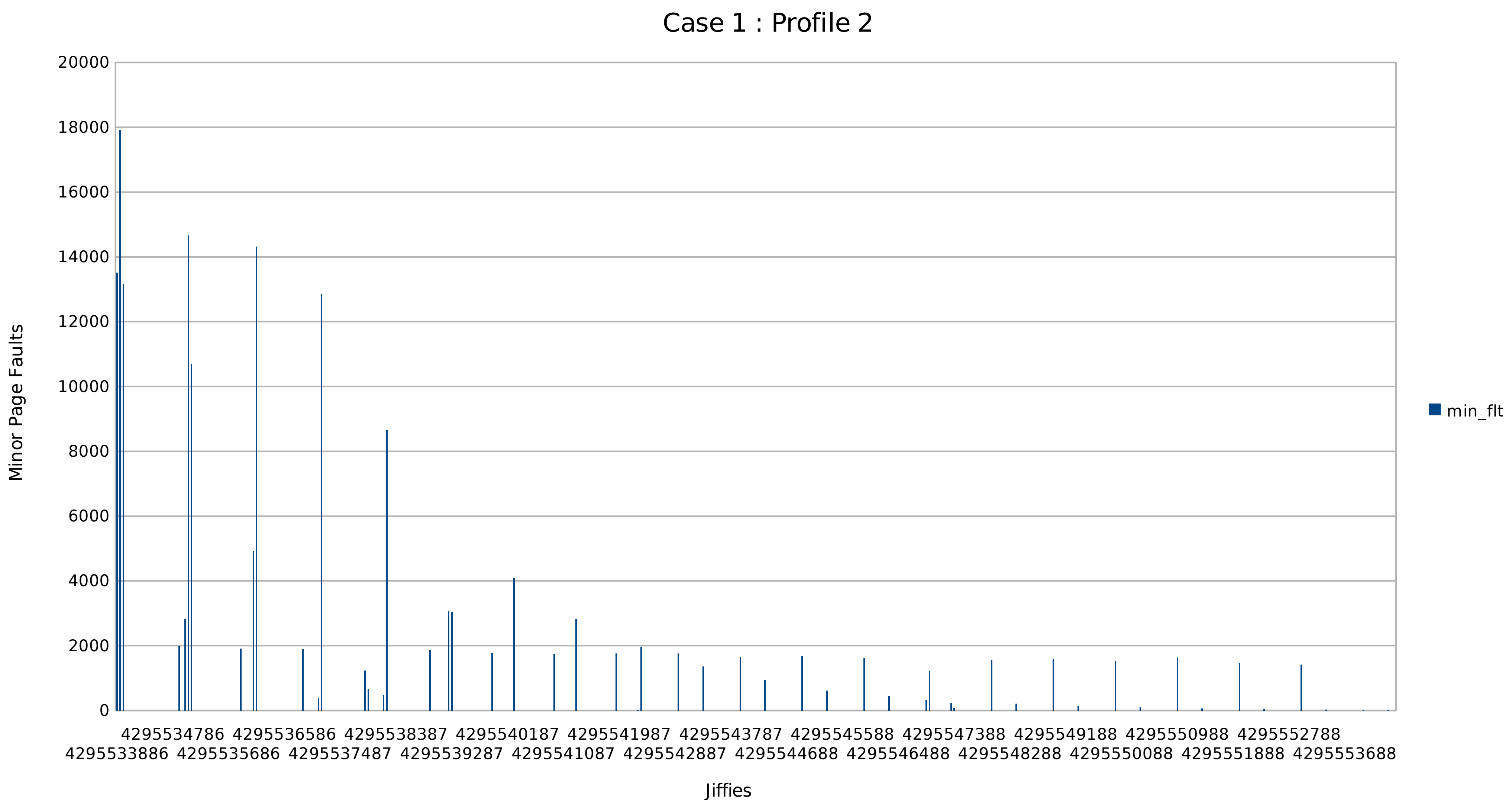
Case Study 1

Profile 1



(x: jiffies, y: minor page faults)

Profile 2



(x: jiffies, y: minor page faults)

Analysis

Observation 1: Neither run, profile1 nor profile2, had any major page faults.

Explanation: Our VMs have close to 2GB of physical memory, with only about 600MB used at the time of the runs. Therefore, there is sufficient physical memory available to satisfy the accumulated virtual address space requirement of the two processes (1GB + a small text and stack) and the kernel does not need to swap out pages. In addition, the memory requests of the two processes do not need disk access to be satisfied; new heap pages can be added from physical memory. For example, no new code pages are accessed while running the process.

Observation 2: At the beginning, page fault rate is high, then quickly jumps to a lower count, and gradually decreases

Explanation: Linux uses demand paging. When the processes call malloc, their memory regions are expanded, but pages are not actually allocated. Only when an address within a specific page is requested, will the page be mapped. Once a page is mapped, it is not evicted unless necessary. Since there is sufficient memory available, pages will not be evicted, so the chances of hitting an address within a page that is not already mapped steadily decreases. At the very beginning of the run it is also necessary to allocate and map second-level page tables.

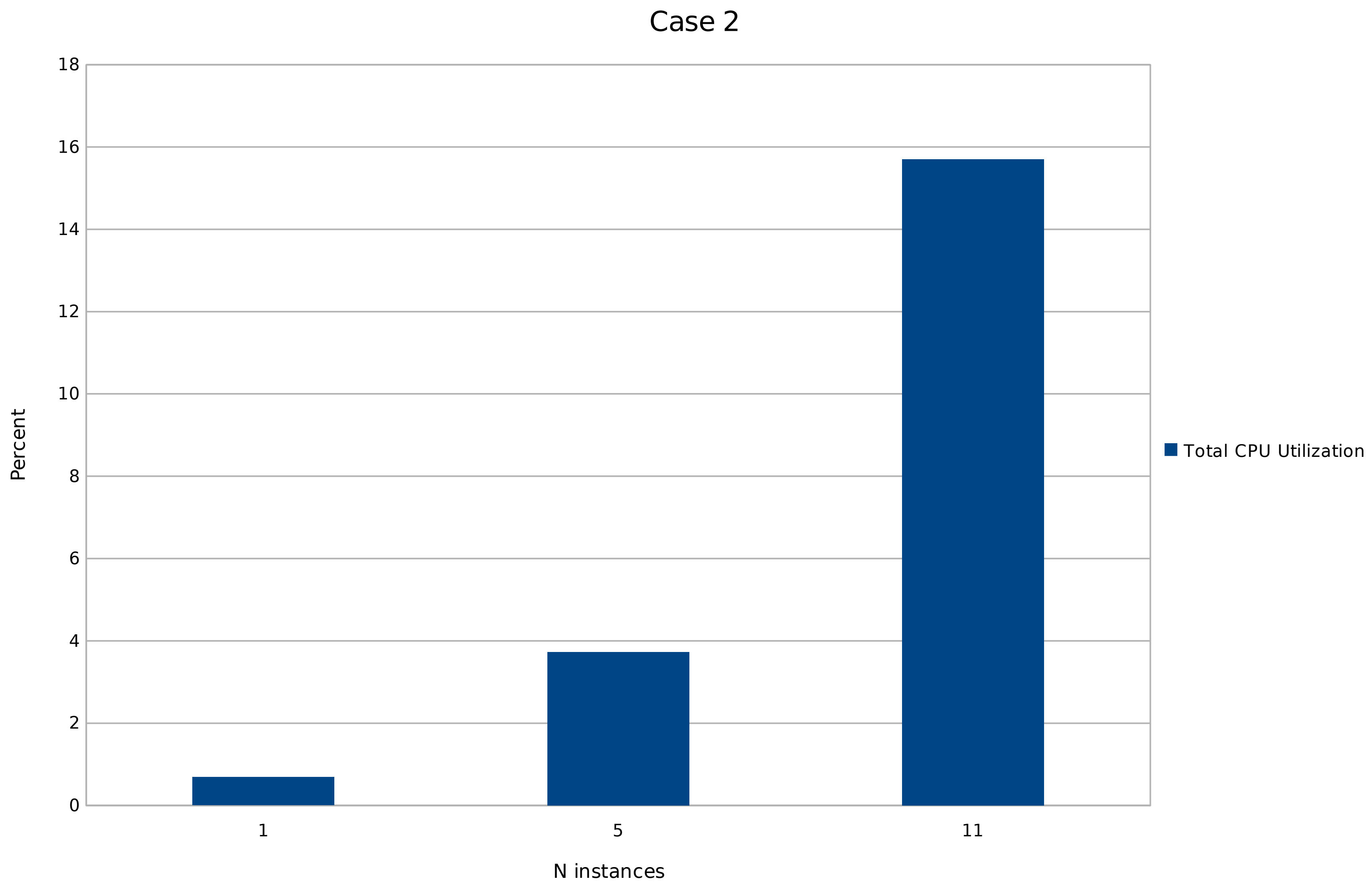
Observation 3: The total page fault count in the second run is lower (~25%) than that of the first run

Explanation: The second process (work process 4) of the second run (profile 2) mostly accesses pages close to the previous memory access (locality-based access). Only in 20% of the cases will it jump to some random address within its heap. Therefore, most memory accesses can be serviced without causing page faults.

Observation 4: The CPU utilization of the second run is lower (~25%) than that of the first run.
(*Note*: the absolute running time is almost the same, since the 20 second sleep time dominates the actual running time)

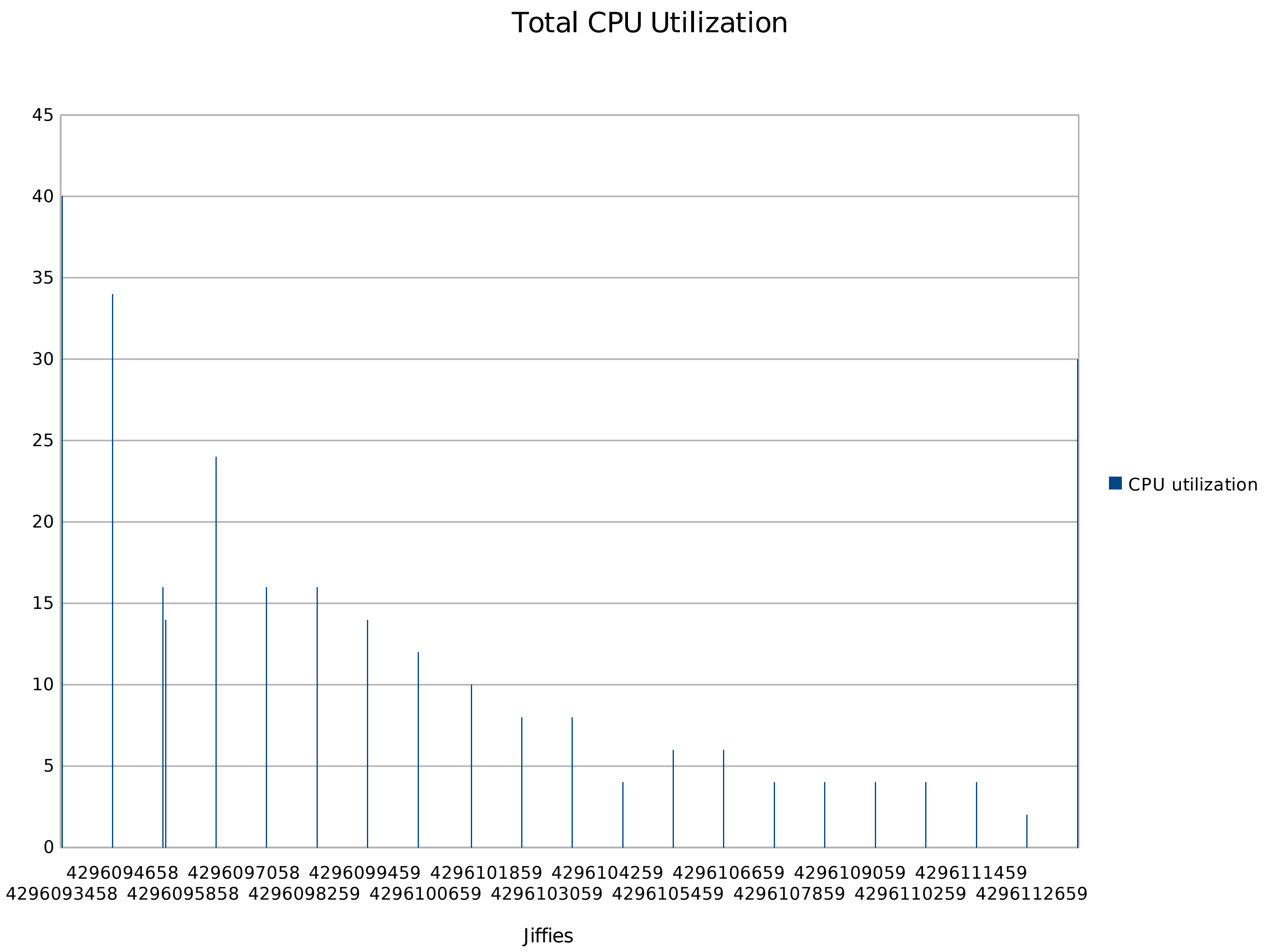
Explanation: The additional page faults cause frequent interruption of the current process to locate an available physical page frame and update the page tables.

Case Study 2
Graph Total CPU Utilization

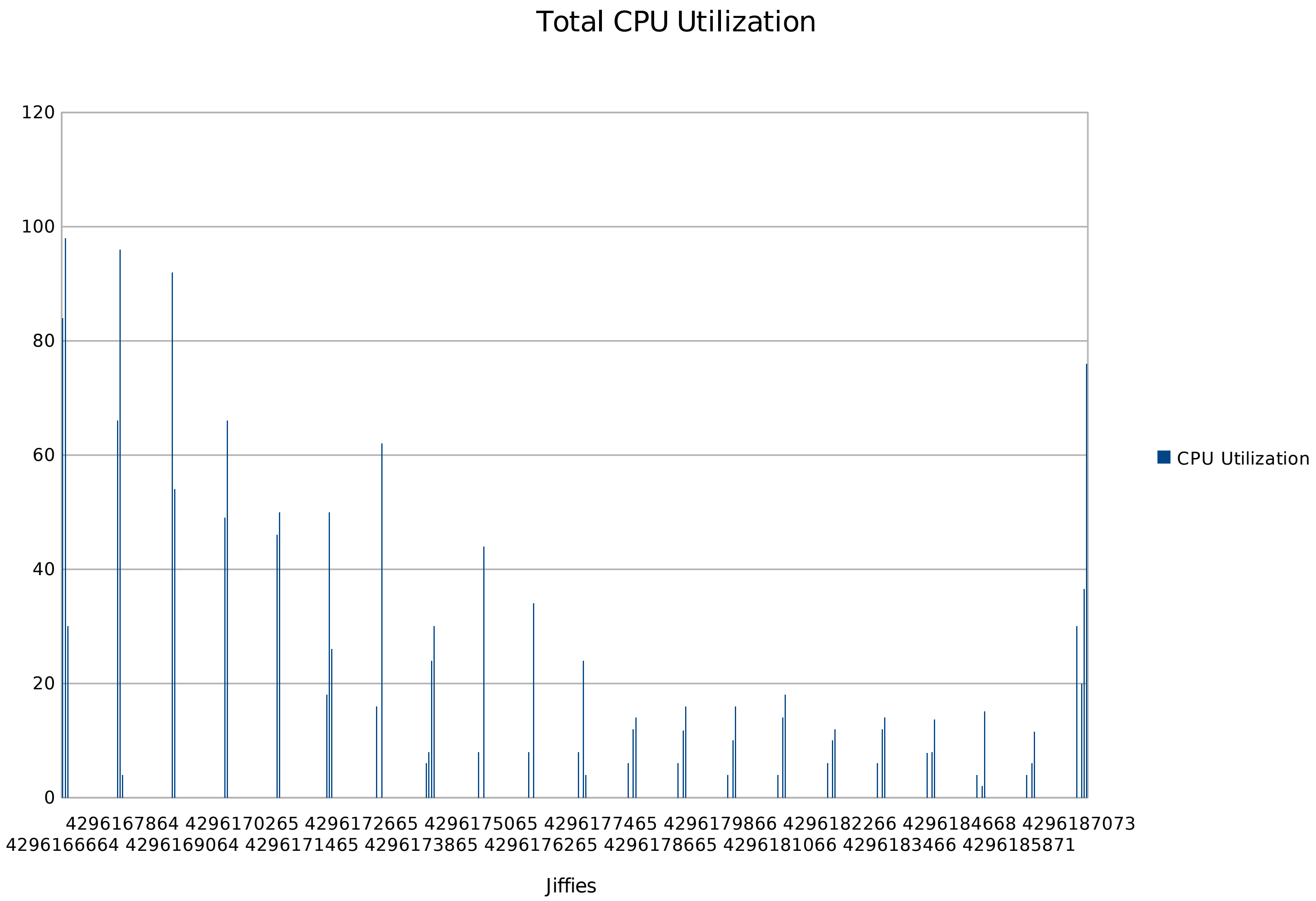


CS 423 – Fall 2011 Group 6: Burrough, Busjaeger, Decker MP3

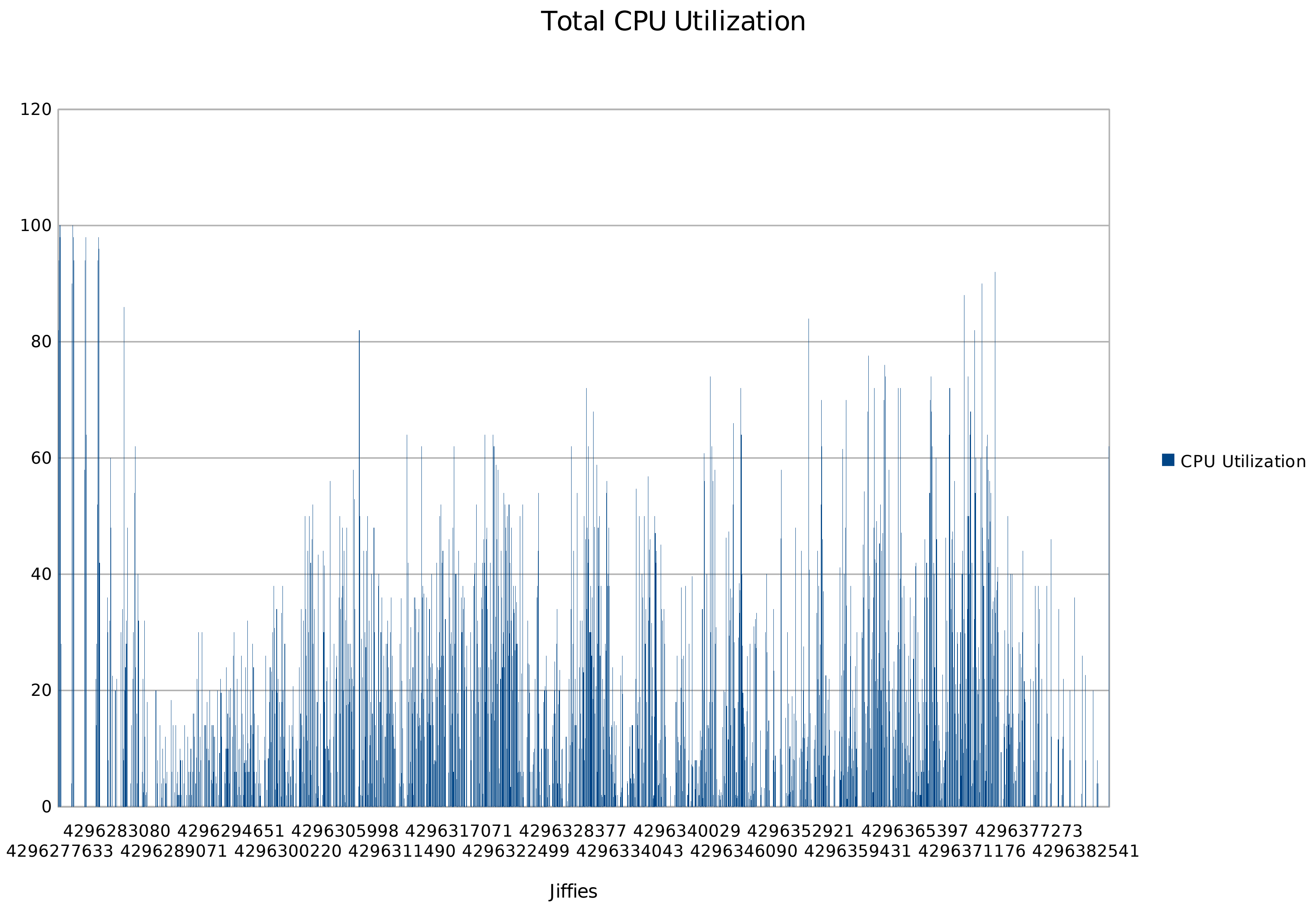
Graph N=1



Graph N = 5



Graph N = 11


Total CPU Utilization
120
100
80
60
40
20
0
4296283080 4296294651 4296305998 4296317071 4296328377 4296340029 4296352921 4296365397 4296377273
4296277633 4296289071 4296300220 4296311490 4296322499 4296334043 4296346090 4296359431 4296371176 4296382541
Jiffies
CPU Utilization

Analysis

Note: For case study 2, we increased the memory used by each work process to 200 MB, so that in the scenario where N = 11, those 11 processes would use up 2.2 GB of memory, which is more than the 2 GB our VM has. In the cases where no major page faults occurred (N = 1, N = 5) is due to the fact that these workloads did not exceed the 2 GB memory limit of our VM. Similar to Observation 1 in Case Study 1.

Averages (total / N)

	Minor page faults	Major page faults	CPU time	Runtime	CPU utilization
N=1	49930	0	140	20151	0.69%
N=5	50132	0	154	4122	0.75%
N=11	57836	3208	1479	9746	1.38%

Observation 1: For the case where N = 1 and N = 5, the same observation can be made as in Case Study 1, Observation 2. At the beginning, the minor page fault count is high and then tapers off.

Explanation: The same explanation for Case Study 1, Observation 2 applies here as well.

Observation 2: Case N = 1 and N = 5 took close to the same time to complete: 20151 vs 20611 msecs. The average runtime per process is therefore much lower for N = 5 (4122 msec). Also, the CPU utilization for N=5 with 3.73% is much better than the one for N=1 with 0.69%.

Explanation: The main time factor in this case study is the second delay between each iteration. Therefore, it is much more efficient to run multiple processes as long as sufficient memory is available. Since at N=5 no major page faults are encountered, the CPU can be utilized much more efficiently.

Observation 3: N=11 shows the highest CPU utilization, but also the highest average CPU time and runtime

Explanation: After a certain period, major page faults start to occur, because the sum of memory needed by the 11 processes exceeds the available free physical memory. Shortly after that, thrashing occurs, because processes compete for memory, constantly awaiting evicted pages to be paged in. At that point processes become disk-bound and most of the work is spent accessing memory, which severely degrades performance. Although total CPU utilization is higher, the majority of time is actually spent doing useless work handling page faults. Each process needs on average 10x more CPU time to complete, so the effective CPU utilization for useful work is lower than with N=5, since 15%/10 = 1.5%.

Observation 4: CPU utilization spikes at the end

Explanation: We believe this is due to process cleanup and termination synchronization

What can be concluded from this case study is that increasing the degree of multiprogramming (from N = 1 to N = 5) is desirable in that overall CPU utilization increases, but that there is a point where increasing that degree (to N = 11) results in thrashing which is undesirable, because it prolongs program execution and wastes CPU cycles.

Case Study 3

The following structure will be placed at the beginning of the shared buffer:

```
unsigned long *write_pos;
unsigned long *read_pos;
int canceled;
mutex_t mutex;
```

Note that in the simplest case, the mutex may be an integer initialized to 0 and both user and kernel space implement the lock function using a test-and-set loop. Busy waiting would be acceptable in the solution presented below, because lock contention is minimal. It may even be acceptable to avoid locking altogether if the three shared variables can be declared volatile. It is also assumed that the kernel module and user application define the constants SIZE (byte size of buffer) and OFFSET (byte size of shared structure) with the same values.

## kernel init

```
buffer_first = vmalloc(SIZE)
buffer_last = buffer_first + SIZE
write_pos = buffer_first + OFFSET
read_pos = buffer_first + OFFSET
canceled = false
mutex = init mutex
```

## kernel thread

```
while (true) {
  sleep(50ms);
  // read profile data
  lock(mutex);
  if (canceled) {
    unlock(mutex);
    break;
  }
  write_pos++=jiffies;
  write_pos++=min;
  write_pos++=maj;
  write_pos=time;
  if (write_pos == buffer_last)
    write_pos = buffer_first + OFFSET;
  else
    write_pos++;
  unlock(mutex);
}

## monitor init
buffer_first = map memory area of SIZE
buffer_last = buffer_first + SIZE;

## monitor thread
current_read_pos;
current_write_pos;
current_canceled;
log_buf;
while (true) {
  sleep(30s);
  lock(mutex);
  current_read_pos = read_pos;
  current_write_pos = write_pos;
  current_canceled = canceled;
  unlock(mutex);
  while (current_read_pos != current_write_pos) {
    log_buf[i++] = *current_read_pos++;
    log_buf[i++] = *current_read_pos++;
    log_buf[i++] = *current_read_pos++;
    log_buf[i++] = *current_read_pos;
    if (current_read_pos == buffer_last)
      current_read_pos = buffer_first + OFFSET;
    else
      current_read_pos++;
  }
  log(log_buf);
  lock(mutex);
  read_pos = current_read_pos;
  unlock(mutex);
  if (current_canceled)
    break;
}
```

The assumption built into this solution is that the profiling thread will never be fast enough to wrap around the cyclic buffer to overwrite entries the monitor has not yet read. In particular, this means the monitor thread has to be started concurrently or shortly after the log thread. If it is possible to share a semaphore between user and kernel space, this problem could be prevented. The kernel would initialize the semaphore to the size of the buffer divided by 16 (bytes per profiling entry) and call 'down' from within the profiling thread after waking up, before acquiring the mutex. The monitor thread would call 'up' after reading a profiling entry. One downside of this approach is that although no data is lost, the log thread may block for a while and report a very coarse-grained profiling entry upon resuming, so precision is lost.

An alternate solution would entail allocating two equally-sized buffers, each sufficiently large to hold 30-seconds of data. One buffer would be mapped into the monitor (called external_buffer) and another would be accessible solely in the kernel module (called internal_buffer). As in our actual MP3 implementation, a work queue function would be used to gather process CPU and fault data, and it would write this data to the internal_buffer.

After the worker function wrote the last data set that would fit in the buffer, it would perform a memory copy operation, copying all values to the external buffer. It is assumed that this memcopy could complete in less time than the monitoring period, though it is not catastrophic if it does not. Since our work queue routine schedules its next occurance at the end of its run, there is no risk of another instance of the routine attempting to modify the buffers. Additionally, at worst, if the copy operation takes longer than the monitor period, the first data point in the next data set will still contain the correct data, the time between the first and second samples will just be slightly longer than average.

After the memcopy completes and the external_buffer has a copy of the data, the monitor usermode process will have access to the data and can use it as needed. A synchronization method is still needed to protect against monitor accessing the data in external_buffer, and the work queue routine copying new data in to the buffer. If the operating system does not have a synchronization primitive for this scenario, another proc file entry could be used as a type of lock. The kernel module could create a second file, which monitor reads continually in one of its threads. The kernel module and monitor could then use it to send signals indicating that they were accessing (and finished) external_buffer.