CS-475 Technical Report: Belady's Anomaly

Alex Kirner

The goal of the operating system is to maximize the degree of multiprogramming, also known as the number of programs your device appears to be running simultaneously. While there are many different angles that must be considered when maximizing multiprogramming, one major one is memory management.

Memory management attempts to fill a number of goals: flexibility, security, size, and speed. Throughout the development of operating systems, different systems have excelled at certain goals, while sometimes wildly failing at others. Today, most operating systems use a system called paging. The idea of paging is pretty simple: a code's memory is divided into small equal sized sections known as "pages". The main memory is also divided into sections of the same size known as "Frames". When a program attempts to access memory at a specific location, the operating system will bring the page containing that location into main memory, placing it in an open frame. A entry containing information about how that page now maps to a specific frame in memory is now saved into a table for future reference.

One really nice thing about paging is that a program attempts to access memory at a "virtual address", which is then translated into the physical address containing that memory. This allows a process to believe it has access to the entirety of the addressable space on a system (so 4GB of memory in a 32 bit system, and much much more in 64 bit systems). This does however cause 1 fatal flaw: If a process can use the entirety of the addressable space, it is possible for a process to use more memory than is actually physically on the computer. This is made even easier when you combine the sum of multiple concurrent processes. This means as we continue to fill up frames in memory, we will eventually hit a point where no open frames are available.

When no frames are available, we call that a Page Fault. When a page fault occurs, the OS must select a frame, and put its page back on disk (where program memory not currently in a frame is stored), and then move the new page to where this old page was just evicted from. But, it is important to choose the right page to remove, as disk access is incredibly slow, so if you evict a frame that then gets used immediately after, you're losing a lot of time as opposed to evicting a frame that won't get used any time in the foreseeable future. Choosing the correct frame is known as an eviction policy, and which one you choose can have a major impact on the performance of your device.

Let's begin with an incredibly simple eviction policy: Random. The name pretty much explains it all: when a frame needs to be evicted, select a random frame, and evict the page in the frame. The random eviction policy can help us see an intuitive relation that exists between the amount of physical memory on a device, and the rate at which page faults occur.

When you attempt to access a frame during a random eviction policy, the odds of that frame causing a page fault is dependent only on the number of page faults since the last time that page was accessed, along with the number of frames in main memory. This is because once a frame is accessed, you can guarantee with 100% certainty it is on main memory after that access. Then for each page fault after that, the odds that the frame is still in main memory after that access is 1/[number of pages in main memory]. Therefore after n page faults, the odds that the frame is still in main memory is $(1/[numFrames])^n$. The only catch for this formula is that it can only be used to look at at the odds that each frame in a set of frames were evicted between during a timeframe where no memory access occurs for any frame in the set, as if an access does exist in that timeframe, then the result of that access affects the odds of all other frames surviving.

However, consider the scenario where frame A is accessed, and there are 10, and then 15 frames in main memory. Then B is accessed (which is on disk in both cases), and then A is accessed again. The odds A is evicted during the B access is 1/10, and 1/15, respectively. However, since in the 15 frame scenario, A is less likely to be evicted, that reduces the chance for A to result in a page fault, which then lowers expected number of page faults between 2 memory accesses of some other page in memory, such as C. So that creates a cascading effect where by decreasing the odds of even 1 page fault, you've decreased the odds of all page faults.

The only other thing to consider when interacting with many frames under this line of thinking is that if in some timeframe, the number of distinct pages accessed is greater than the number of frames in memory, a page fault will be guaranteed. However, by ignoring that final memory access (meaning that 1 frame isn't accessed), we can show that the chance that no page faults during the new reduced timeframe has increased. And by having the page fault occur the furthest down the line, it again cascades and decreases the chance. Additionally, the odds that this scenario even occurs goes down as we increase the number of frames.

What this means is that if we increase the amount of memory in RAM while using a random eviction policy, it will cause a faster run time. This also intuitively makes sense since if memory was large enough that all pages could be stored in main memory, there would be zero page faults, so it would only make sense that the function graphing the relationship of memory and runtime would be continuous between those 2 points.

However, this is not always the case. In 1969, computer scientists at IBM: L.A. Belady, R,A. Nelson, and G.S. Shedler published a paper entitled: "An anomaly in space-time characteristics of certain programs in a paging machine", which explained how they had discovered an exception to the aforementioned relationship.

Belady and his team were using an eviction policy known as FIFO. In a FIFO policy, you simply evict the frame that has been in main memory the longest. Belady's

team was doing testing on a FIFO queue, and noticed a strange occurrence: When increasing the allocated memory for certain programs, the runtime would actually decrease. They give the example of a program that has 5 pages with the following memory accesses. 123412512345. When running with 3 allocated frames, the program had 9 page faults. However, upon increasing the number of frames to 4, the number of page faults increased to 10.

Their immediate reaction was to question how this occurred, since the relation between memory and runtime held for everything else. One important thing to note is that my explanation for why this works for random relies on statistics. Increasing the amount of memory, decreases the amount of expected page faults. Technically speaking, if this anomaly occurs when certain programs are run using FIFO, the anomaly could also occur when that same program is run using random, and the random elements selected happen to mimic FIFO (at least enough to cause the anomaly). However, this is a bigger problem with FIFO as it only requires 1 condition to be true (the memory access order must follow certain properties), while random requires both the program to meet those properties and for the evicted frames to occur in a very specific order. Belady clarifies this by explaining some conditions: “1. The algorithm must be the repeated application of a rule for selection which is independent of the number of times it has been invoked. 2. There can be no special symbols or special positions in a reference string or special relationships of symbols and positions which influence its behavior. 3. Its selection must be independent of the relative store sizes: it may not have information that it is being applied to the larger or smaller of two stores or, indeed, that there exists a store of any size other than the size of the store to which it is being applied.” (Belady, 351).

The goal of these conditions is to rule out cases in which a combination of the memory access string and the eviction policy combine to result in cases where the anomaly occurs where it shouldn’t. For example, FIFO violates condition 1 since each time you run it, the result will be different. You could run code that isn’t even susceptible to Belady’s anomaly, and increasing the memory could slow runtime purely just because it selected good frames to evict on the smaller memory size, and worse frames to evict on the larger memory size. Condition 3 effectively says that you’re not allowed to manipulate the memory access in order to cause or not cause the anomaly. Similarly, condition 3 covers the case where if you were to manipulate the algorithm so that it just left frames unused, that would result in a technically bigger main memory appearing smaller. Another way to fix this condition would be to define belady’s anomaly as an inverse relationship between *utilized* memory, and runtime. This also accounts for the fact that if you had a terabyte of RAM on a computer, and your total usage never got anywhere near that, increasing to 2TB would have zero effect.

Belady goes on to explain a number of specific scenarios that would result in the anomaly occurring during the FIFO eviction policy.

Cluster Rule

Let there be 2 computers, one with S frames in main memory, the other with L frames. Additionally, let A be some arbitrary integer such that S < L < A < 2S.

In this case, bellady says that the concatenation of these sequences will result in the anomaly

1) 1, 2, …., L
2) 1, 2, …, (L-S+1)
3) (l+1), (l+2), … a
4) Sequence #2 again
5) (L-S+2), (L-S+3), …, (2L - A + 1)
6) Sequence 3 again

To explain this, we first should define a Compulsory Miss. A compulsory miss is a miss that occurs when a memory location is accessed for this first time during a program. We know this will miss no matter what, because it cannot be in a frame unless it was accessed previously.

Now to look at the number of page faults

1) During string 1, we are accessing the first L data values for the first time, so both computers will have L page faults. However the L memory will populate each frame exactly once without evicting, while the S frame will evict the first (L-S) frames.
2) During string 2, the L sized memory contains all the pages needing to be accessed, so it won't page fault, nor will it evict anything. The S memory will begin at 1 again, which doesn't exist, evicting the oldest frame, which will be the page with the next smallest number. This will repeat, resulting in L-S+1 page faults
3) During step 3, both frames are seeing another (A-L) new element, meaning A-L additional page faults for both. However, since S< L < A < 2S, we know that A-L < S. In the L memory, this will start evicting the smallest A-L values (as those are the oldest ones from step 1), while in the S memory, this will not evict anything from step 2, as A-L + (L-S +1) = A-S + 1 ≤ S.
4) In the L memory, we are starting at 1 again, which has been evicted in step 3. As we continue to add higher numbers, it's evicting things even further ahead, resulting in every access being a page fault, giving us L-S +1 page faults. As for the S memory, we said all the accesses from step 2 still existed, and since step 4 is a repeat of step 2, no page faults or evictions will occur. This puts the number of page faults in each category even again
5) If we look at sequence 4 for the L memory, we know that it started by evicting elements of sequence 5 in order. Since sequence 4 has at least 1 element, this means that each element of 5 was evicted either during 4, or earlier in 5. This gives us another 2l - a + 1 - (l-s+2) +1 elements evicted. As for the S memory, It

contains only elements in 2 and 3, which leaves a gap between L-S+1, and L+1. Since we start at L-S+2, we know that it will begin evicting frames from L-S+1, resulting in only page faults, meaning that the number of page faults in S is the same as L still

6) For the L memory, we know for a fact that it does not have enough frames to hold steps 3-5 all in memory. Therefore as it started evicting elements from 3, it would evict at least the first frame in 3, meaning that step 6 will only hit page faults. For the S memory, it will only hit a page fault if it has started evicting the frames from step 3. Step 4 evicted no frames, and we know that step 5 evicted 2L-A+1-(L-S+2)+1 = L-A+S frames. Since step 3 had a-l frames added, and L-A+S - (A-L) = S, we know it did not evict any frames from step 3, and therefore step 6 will not hit any page faults

This gives us a result where the L memory had A-L more frames evicted than the S memory. Now this feels like a hyper-specific set of data, and to some extent it is. However, we must remember that memory access is not random. For example, arrays are a very common data structure, and arrays are a collection of successive data in memory. So while this exact set of conditions may be somewhat rare, it still can occur quite a bit. Additionally, it's important to remember that while all the data in the example were somewhat close together, you could almost think of the numbers themselves as variables pointing to completely different memory locations. As long as every time you access whatever location "1" is, you access the same location. This could begin looking like a set of functions being called. Especially as S,L, and A get larger, you could see major loss in efficiency.

The other method that I won't go into too much is the cyclic rule. The Cyclic rule states that when $s < l < 2s-1$, and $a \geq l + s/2$, then if you take the string S, S+1, S+2, …l, 1, 2, …, s-1. And then follow it up with elements alternating from substring (1): s, s+1, s+2, …, a, 1, 2, …, s-1. And (2): 1, 2, 3, …, a. Then you will also encounter the anomaly. This one is significantly less common due to how much jumping it does, though it's possible to encounter it when handling pointers in something such as a linked list.

Ultimately, while Belady's Anomaly only occurs during very specific scenarios, the nature of how computer memory access works makes those specific scenarios more common than one might think.

Ultimately, FIFO is suboptimal not only because of Belady's anomaly, but also because it ignores the Principle of locality, which effectively says that data accessed in memory is more likely to be either the same or close to data very recently accessed in memory. Other methods such as Least Recently Used and the CLOCK algorithm, which is an

approximation of LRU, do a much better job at adhering to that principle than FIFO, meaning they are more efficient usually, even when Belady's anomaly is on the table.

What Belady's anomaly taught the researchers was that they needed to come up with a better algorithm. This eventually resulted in the proposal of Belady's Min, which is a provably optimal eviction policy where you evict the frame that will be used the least soon in the future (if A will be used in 10 memory accesses, and B will be used in 20, evict B). However, computers cannot read ahead, which led to the creation of LRU. LRU works by evicting the least recently accessed frame. When that was found to be too difficult to implement in a fast way, the CLOCK algorithm, which is an approximation of LRU which evicts *a* frame that hasn't been used recently. We currently use CLOCK in modern computers. So all this came from some researchers who ran a test and noticed something that went against their intuition.