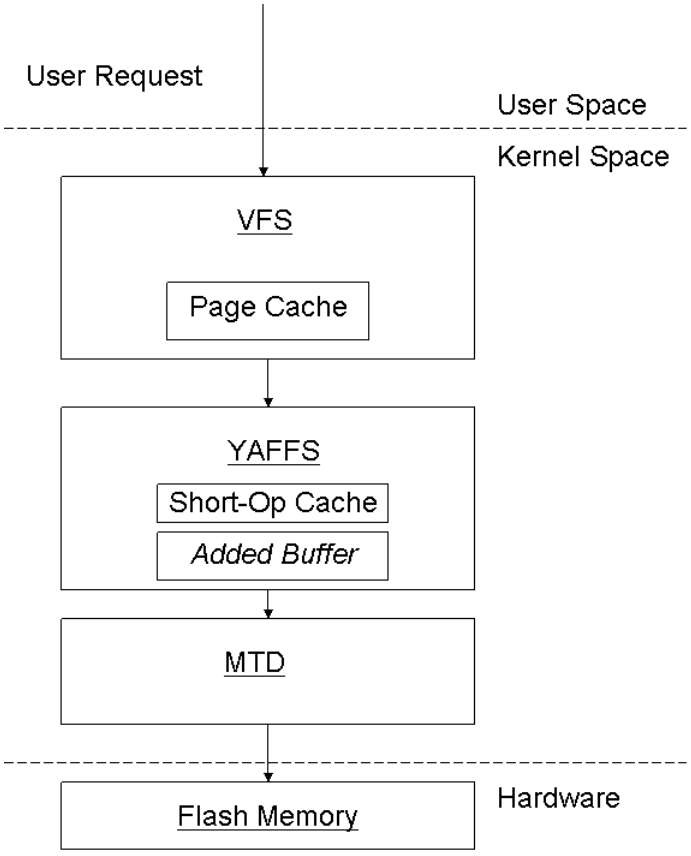
The Effects on Read Performance from the Addition of a Long Term Read Buffer to YAFFS2

Sam Neubardt

My research project examined the effects on read performance from the addition of a long term read buffer to YAFFS2. YAFFS (Yet Another Flash Filing System) is a file system, the program that organizes data on a storage device (like a hard drive), that is specialized for flash memory. Flash memory is a type of storage device that is typically found in embedded devices, like cell phones or PDA's; USB "thumb drives" also contain flash memory. Hard drives and flash memory have several key differences. Flash memory is smaller and uses less power than a hard drive, making it a popular choice for systems that run off a battery and must conserve power usage. Data on a storage device is organized into pieces called sectors or pages. This is done to allow data to be changed without having to modify everything on the drive. Hard drives have much smaller sectors than flash memory, which does give them a slight advantage in performance. While the data on a hard drive can be changed many times without causing physical wear on the drive, the data on a flash memory device can only be changed a certain amount of times (often over 500,000 times per page, but it still adds up) before the reliability of the drive starts to corrode. YAFFS is typically run on the Linux operating system, as was this experiment.



Hierarchy of major buffers

Caching (here equivalent with buffering) is the process of storing data so it may be quickly accessed. Buffers are stored in RAM because accessing data from RAM is much faster than

retrieving data from a storage device. Caching is an important part of all file systems, especially for those intended for use on flash memory, as it can greatly improve the speed at which data is read off the drive. As flash memory is primarily used in embedded devices, where power consumption and thus battery life is of great concern, reduced disk access can result in improved battery life. Since access times from RAM are shorter than those of flash memory, an overall speedup in terms of read operations also occurs when data is read from RAM rather than from flash.

YAFFS contains a short term buffer, called the "short operations cache," that is used to group small input and output operations together (reducing wear and improving performance), but has no facilities for long term caching. A long term cache would be larger and would store data that is requested frequently for a longer period of time. The greatest performance increase would be seen from caching the most frequently accessed data, which would then be held in the cache for a longer (if not indefinite) period of time, rather than temporarily as in the short operations cache.

The Linux kernel provides a data cache of its own called the page cache. When a request for data from a file is made, the kernel first checks the page cache before initiating an I/O (Input/Output) operation. If current data is found in the cache, it is returned; if not, the data is read from the device in which it resides and inserted into the cache. This cache exists above the file system layer; if a request is satisfied from the page cache, the file system may never receive the request.

It was hypothesized that adding a long term cache would lead to increased. Simulated flash hardware was used to test this hypothesis. Actual hardware was not used to avoid the irregularities that occur between different vendors and models. Although access times between RAM and flash differ, the flash simulation code adds a delay whenever the simulated device is accessed, to reflect the difference in performance and to better emulate real-world access speeds. The experiment consisted of two test groups, one with and one without the added long term buffer, and two tests were performed per test group. A program for benchmarking file system performance, called Iozone, was used to measure and record the performance of the two test groups. The test consisted of reading and then re-reading a set

of data from the drive. The first time the data was read, it had to be read from the disk, since data would be added to the cache after it had been read at least once (caching data before it has been explicitly accessed is called preemptive caching, but was beyond the scope of this research). However, the second read offered the opportunity for a cache hit. Contingent on the data being fetched from the cache, the second speed should have been faster than the first.

Figure One
Read Test (Record Length = 8 Kb)

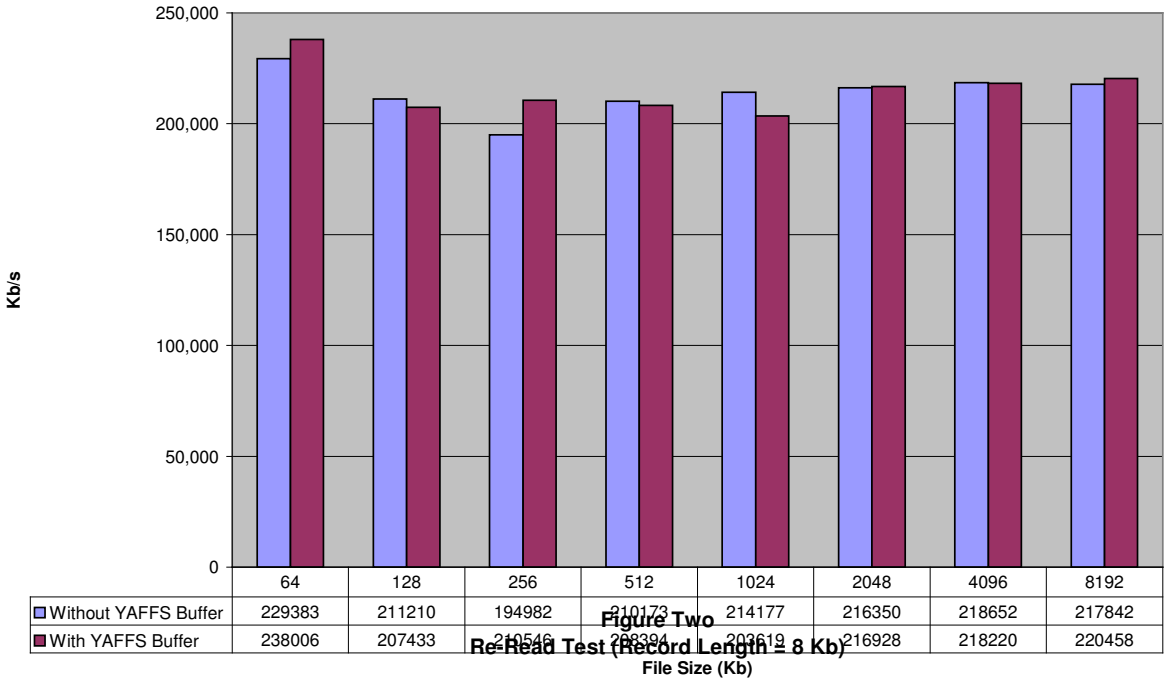
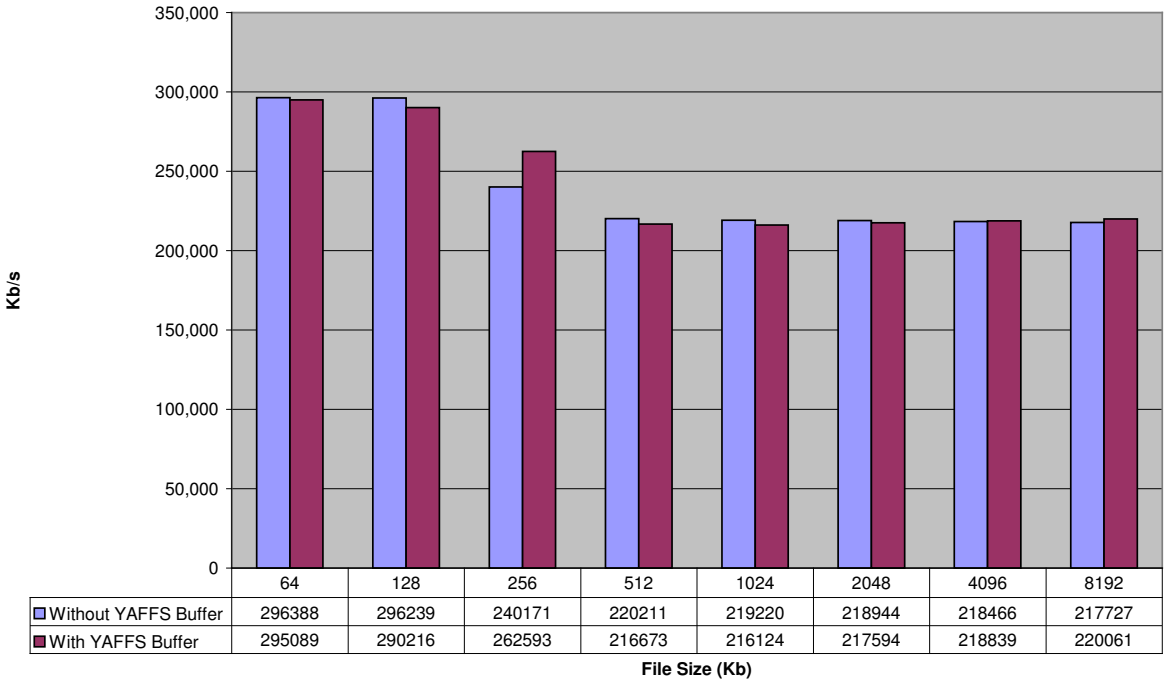


Figure Two
Re-Read Test (Record Length = 8 Kb)



The Read Test (Figure One) established the baseline for both versions of YAFFS. Since it was the first time the test file was requested, the data came from the flash drive. Figure Two ("Re-Read Test") demonstrated the effect of buffering. Throughput for smaller files jumps higher in the Re-Read test in both cases, showing that versions both with and without the additional buffer are experiencing buffering from some source, most likely the Linux page cache. If this buffer were not present, the unmodified versions of YAFFS would not experience the increase in throughput shown. The negligible difference in throughput between the unmodified and modified versions of YAFFS remained consistent throughout the tests.

The results did not support the hypothesis, in that the addition of complementary buffer to YAFFS2 did not lead to improved read performance. The Linux kernel did a competent job buffering data in this case which made the added cache unnecessary. Although the addition of a cache did not lead to any usable improvements, the data could be used to better tune both YAFFS and the Linux page cache.

I've been interested in computers since my family got our first one when I was five. When I entered my schools science research program in the 10th grade a topic involving computers seemed like the logical choice. I was interested in Linux and explored different possibilities using it, ultimately deciding on file systems.

I performed the research in my room at home. My mentor, Charles Manning (who wrote YAFFS) lives in New Zealand so face to face collaboration was not an option and correspondence was limited to email. However, since no actual lab was required, the separation was not the hindrance it would be in a more physically applied discipline. Summer weather, coupled with the extra heat generated from the computer, proved to cause more of a hindrance, limiting work to the morning and evening when outside temperatures were cooler. Since my mom doesn't like air conditioning, much of the research was performed while furiously sweating over a keyboard. The want to escape the brutal

heat might have ended up being the biggest immediate motivator.

Although computer science is not entirely composed of math, it has its roots in the mathematics of computation, and that carries into the modern discipline. I learned more mathematics in learning the background computer science theory for the research than in preparing for and conducting the actual research. I am not, and have never been a strong student in math and was relieved to find that the math required for most of the programming I would be doing was nothing too extreme.

The paradigms and methods of thinking associated with mathematics are often more important than applied math. Knowing how to find the imaginary roots of a complex equation is important in certain fields, but for most people just knowing how to process information and think algorithmically proves to be more practical. The amount applied math used while programming varies from project to project. Just as in mathematics, programming requires one to think a few steps in advance to be able to solve a problem. In more complex and higher level math, solutions may not be entirely linear, which makes it important to think ahead a few steps. Both math and programming are open ended exercises, requiring forethought before execution. In that respect, studying math is beneficial even if it is not one's primary focus. Examining the work of great mathematicians is akin to examining the work of a brilliant chemist, writer or athlete. Programming and the research process made math more alive for me because I saw the mathematical process at work in areas of greater interest to me. After I started programming, both my aptitude for and grades in math improved.

Although I had experience programming before starting the project, I was not familiar with low level and operating system programming, the areas in which most of the programming work for the project was based. Before beginning to write code of my own, I first tried to acclimate myself with the existing YAFFS code base, which I would be modifying.

Hashing is the process of reducing data to a small number. It’s used for several different things in computer science, two of the most common (both of which are implemented in YAFFS) are error correction and data structures. Because the data stored on flash memory can corrode, a hash called an

error correcting code (ECC) is stored by YAFFS. Using the ECC, the file system can detect and correct small errors in data. Certain data structures (variables which are organized to enhance their function) make use of hashes. One such structure, the hash table, was directly used in this study. The added buffer was constructed using a hash table, which allowed data to be quickly stored and retrieved. A hash table takes a hash of the data that is to be stored and uses that hash to organize the data within a list. Hashes make both of these applications possible by allowing the computer to process data more quickly by processing less of it.

Numbers are represented using different bases. Most numbers we see are base 10, they have 10 possible coefficients (0-9) per digit. When numbers are represented in different bases, the principal is the same, just applied differently. When we're taught arithmetic, teachers often have us break the numbers into different places, like the one's place, the ten's place etc. These places are really the base of the number raised to a certain exponent, increasing with the length of the number. For example, a 3 digit number in base 10 has three places, (reading from left to right, left being largest) 10^2, 10^1, 10^0, or 100, 10 and 1. To construct a number from these bases, we use the values 0-9 as the coefficients for the digits. So the base 10 number 738 would be $(7 * 10^2) + (3 * 10^1) + (8 * 10^0)$ or 700 + 30 + 8 = 738. Since we deal with base 10 numbers every day, and are used to looking at them, we don't actually multiply out everything to understand what the value of a number is; base 10 appears “natural” to us, maybe because we count on 10 fingers. However, even though base 10 numbers are convenient for us, they are inconvenient for computers to process.

Computers are essentially large arrays of transistors, or switches. When a computer manipulates or processes data it is, at the most basic level, manipulating tiny switches, activated by electrical impulses. This is true when a computer sends an email, plays a song or performs any operation. These operations happen very quickly, by many transistors, which allow computers to perform higher level tasks and be useful to society. Since computers are composed of switches with two states (on and off), it is efficient for them to represent numbers using two states as well, by

representing them in base 2 or binary.

Binary numbers work the same way base 10 numbers work, except the available coefficients are either 0 or 1. The two possible coefficients represent the two states of a switch, with zero representing off and 1 representing on (think of the symbol used to denote on and off on some switches, the 1 symbol is on and the 0 symbol is off). The process of constructing a number in binary form is the same as it is for base 10 numbers. For example, the number 23 (base 10) is 10111 in binary, or $(1 * 2^4) + (0 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)$ or $16 + 0 + 4 + 2 + 1 = 23$. Each binary digit is called a bit and 8 bits are called a byte.

Binary isn't just useful for computers, we can use it to count to 31 on only one hand. On your right hand, extend all five fingers and point your palm towards your face. Starting from your righter-most finger (the thumb), assign each finger a value starting with 2^0, increasing the exponent by one for each finger, (the pinkie representing 2^4 or 16 and the thumb representing 2^0 or 1). Try writing the value for each finger on the finger to help you remember at first: 16 (pinkie), 8, 4, 2, 1 (thumb). If a finger is extended, it indicates that that place has a value of 1, the bit is activated, if it is lowered, the bit is off and its value is zero. Now bring all your fingers down, like a fist, to denote zero. To count to 1, raise your thumb; for two, lower the thumb and raise the index finger; and, for three raise the thumb while leaving the index finger extended. Add the values of all extended fingers to find the total sum shown on your hand. If you keep continuing in this pattern you can reach 31 on just one hand.

Although binary is the most efficient form for computers, it is difficult for humans to quickly parse. That's why yet another base is used when dealing with computers, called hexadecimal or base 16 (hexa = 6, dec = 10). Many things with computers are powers of two, like the values of bits, and 16 is a power of 2 as well. Since each digit of a binary number holds a smaller value, they are often longer than numbers represented in higher bases. Although hexadecimal numbers are not as easy for humans to deal with as base 10 numbers are, it is still easier to look at a short hexadecimal number than a long binary number. Hexadecimal digits have possible coefficients ranging from 0 to 15. The letters A

through F represent the numbers from 10 to 15 (if letters weren't used, there could be more than one digit per place which would make things even more confusing). To differentiate hexadecimal numbers from decimal or other numbers, hexadecimals are often prefixed with either $ or 0x. For example the number 4F in hexadecimal is 79 in decimal: $(4 * 16^1) + (15 * 16^0)$ or (4* 16) + (15 * 1) = 79.

Bitwise operations are logical operations performed on a number or variable that manipulate that variable at the bit level. Bitwise operations mimic the structure of a computer's processor. In a processor, transistors are arranged to form logic gates, pathways where the output (still 1 or 0) depends on the input. For example, let's compare the binary numbers 110 and 100 using the bitwise OR operation. The OR operation sets the result to true (1) if one or both of the inputs are set to true and false (0) when both inputs are false. To compare those two numbers, corresponding digits are compared with the OR operation: 110 OR 100 is (1 OR 1), (1 OR 0), (0 OR 0), resulting in 110. Bitwise operations are very useful for low level data manipulation. On some processors it is also faster to use bitwise operations to multiply or divide rather than using the arithmetic operators. YAFFS makes extensive use of bitwise operations, partially in part to communicate with flash hardware at a low level.

The most crucial part of research (or anything really) is persistence. It took two hot summers filled with frustration to even collect data – data that showed the project hadn't worked as expected. At first it was disappointing to learn that all that work didn't even lead to a slight improvement, but just because the hypothesis wasn't proved true doesn't mean that a project is worthless. Just as much can be inferred from failure as from success. Confidence and persistence are all you need to succeed.