Effectiveness of RAID Level 0

A Performance Comparison of Software-Simulated RAID 0 Configurations

Stuart A. Miller
Department of Computer Science
Missouri University of Science and Technology
Rolla, MO
sm67c@mst.edu

***Abstract*—RAID 0 provides a significant performance increase for large data storage systems. Ideally, the performance increases a rate directly proportional to the number of drives in the configuration. This can be performed by simulating both best case and average case RAID configurations in software.**

Keywords—RAID; Level 0; Performance; Complexity; Striping; Simulation

I. INTRODUCTION

Originally proposed in 1987 by David A Patterson, Garth Gibson, and Randy H Katz of the University of California-Berkeley in their paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)", RAID provides a means for improvements for data storage in terms of increases in both performance and reliability [1].

Through the use of multiple easily-replicable disks, a RAID configuration is firstly able to achieve increases in performance by distributing data throughout all the entire arrays of disks. This is achieved by taking data that is spatially local and distributing it across multiple disks. This allows for consecutive sections of data to be independently and simultaneously read from each disk, creating a substantial decrease in the time is takes to read large sections of data.

Secondly a RAID able to provide increased reliability through data redundancy. This is achieved through either duplication of data, or through the calculation of one or more parity bits that store the state of the data in the array. Upon writing data to the array, it is either copied to the mirror drive of used to calculate a parity bit stored on a member drive. Should a drive fail, it can easily be replicated either from its duplicate drive, or recalculated from its parity bit and the surrounding data.

RAID 0 is the most basic level of RAID storage. While RAID 0 provides no data redundancy, it is perhaps the most simplistic configuration and provides excellent opportunity for performance increase. Data stored in a RAID 0 configuration is distributed or "striped" across all the disks in such a manner as to consecutively distribute it throughout the array. Because the data can be accessed in parallel, an array of *n* drives can theoretically provide a data read/write rate up to *n* times faster than that of a single disk.

Figure I shows an example of a level 0 RAID containing four drives. As you can see, the series of data contained in blocks d1-d4 can be read simultaneously. Doing so provides a read/write speed increase of four times.

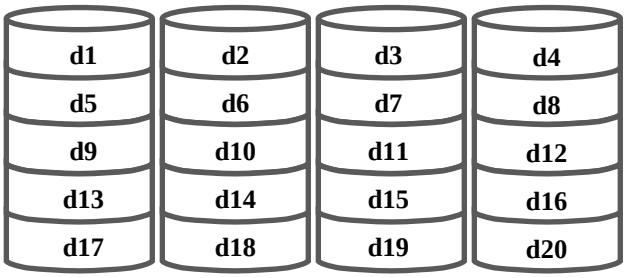

Figure 1: A RAID 0 configuration with four drives

It is worth noting that a RAID 0 setup provides absolutely no data redundancy. Should one drive fail, the entire array will consequently fail. The loss of this single drive causes the entire data set to be discontinuous and therefore unusable.

II. SIMULATED DRIVE OBJECTS

In order to run RAID read/write simulations in software, a system for simulating drives was first implemented. The end goal was to create a single executable with the ability to simulate a series of disk accesses. For the purpose of this test, a disk access was defined as either a read or a write, as it makes no difference to the RAID functionality. The simulation script was written in C++ because of its object-oriented properties. This allows for multiple disk objects to be created and destroyed while retaining code clarity and readability.

Data structures for both disks and arrays of disks were created to facilitate this.

A. Disk Objects

The basic disk object was written as a class with its own library of functions. Set and get functions were extended down

to the disk level in this class. This allowed for the low level writing of an individual byte at a time to a disk. Included in the write to disk function was an access time modifier. This caused the program to wait while for a set amount of time to simulate the head reading a byte of data. This was, however, set to 0 for the tests as in this RAID configuration all the data read was consecutive on the block level.

Each disk's data consisted of a large array of unsigned 8-bit integers representing bytes stored on a hard drive. Data types were customized for this and for many other variables so as not to take up excessive space. In this case, unsigned chars were typecasted as *uint8* types to accommodate this. Additionally, the total size of the data storage arrays was configurable and set at runtime. For the purposes of this test disks size was kept consistent. This is logical as all disks in a RAID configuration must be of the same size to accommodate even striping.

In order to accommodate the block separation required for a RAID configuration, each disk was partitionable. This was achieved by creating an array of partition indices stored inside the disk data structure. A function to assign disk partitions was created. Upon calling the partition function, the partition index array was populated based upon the passed partition size. Upon later calls to access disk partitions, this array was referenced and used to calculate a base address.

B. Disk Array Objects

A disk array type was also created to emulate the functionality of an entire RAID setup. A disk array contained the appropriate disks and several associated access variables.

Each individual disk was referenced by a pointer stored in an array within each disk array type. Pointers provided the best method to quickly access down to the data level with the least overhead possible. In this manner, functions at the main program level could simply follow a chain of pointers directly down to the data the calling function needed to access.

Additionally, an array-level block index was calculated. This consisted of a two dimensional array containing first a disk number and then a partition number. Upon initialization of the array type, a function rotated through each disk's partition index array and sequentially copying each into the block index array. Doing so created an array level index similar to that seen in Figure I.

A library of functions was also created for array level access. In addition to the constructor and initialization functions, a write function was also written. This function took a block number as an input and calculated the appropriate disk and partition with which to call the disk level write function. For the purposes of this test data was written a block at a time. This means that individual byte accesses were physically consecutive and would incur the least read/write time possible. However, the access time was still accounted for prior to each block access. This simulates the time that the read head would take to physically move to the track on which the data is stored and then wait for the rotational delay of the disk. This number was configurable for each test run. Figure II shows some common access time for hard drives on the consumer market [2]. The average value of ~14ms was chosen for the simulated access time. This value was kept consistent across all trial runs.

Seagate Laptop Thin SSHD	2.21	Seagate Momentus 7200.4	13.33
Western Digital Scorpio Black WD3200BEKT	7.25	Hitachi Travelstar 7K750	15.45
Western Digital Scorpio Black WD5000BEKT	7.31	HGST Travelstar 7K1000	15.69
Toshiba MK5061GSYB	7.69	Samsung Spinpoint M8 HN-500MBB	15.83
Toshiba MK6465GSX	7.72	Western Digital Scorpio Black WD7500BPKT	16.10
Western Digital Scorpio WD1200BEVS	7.97	Seagate Momentus	16.24
Toshiba MK6461GSYN	8.00	Western Digital Scorpio Blue WD5000LPVT	17.27
Hitachi Travelstar 7K320	8.09	Western Digital Scorpio Blue WD10JPVT	17.86
Samsung Spinpoint M7 HM500JI	8.20	Western Digital Red WD10JFCX	18.10
Fujitsu MJA2500BH	8.31	Hitachi Travelstar 5K1000	18.35
Western Digital Scorpio Blue WD3200BEVT	8.53	Western Digital Scorpio Blue WD10SPCX	18.42
Samsung Spinpoint MP4 HM640	8.62	HGST Travelstar 5K1500	18.95
Toshiba MK5056GSY	8.63	Samsung Spinpoint M8 HN-M101MBB	19.46
Western Digital Scorpio WD1600BEVS	8.77	Seagate Momentus XT	20.01
Hitachi Travelstar 7K500	8.86	Toshiba MQ02ABF100	20.47
Samsung Spinpoint M7E HM641	8.88	Hitachi Travelstar 5K750	20.74
Seagate Momentus XT	9.03	Toshiba MQ01ABD100	21.29
Western Digital Scorpio Blue WD5000BEVT	9.18	Toshiba MQ01ABF050H	21.44
Hitachi Travelstar 5K500.B	9.42	Toshiba MQ01ABD100H	21.69
Toshiba MK5055GSX	9.44	Toshiba MK7559GSXP	22.60
Fujitsu MJA2500CH	10.35	Western Digital Scorpio Blue WD7500BPVT	22.73
Seagate Momentus 5400 FDE.2	11.02	Toshiba MK1059GSM	24.29
Seagate Momentus 5400	12.20	Seagate Laptop Ultrathin HDD	25.05
Samsung Spinpoint M6 HM500L	12.28	Seagate Momentus 5400 FDE.3	26.07

Average: 14.07ms

Figure II: A table of hard drive access times [2]

C. Interfacing

The object-oriented approach provided for a clean and easy to understand simulation. Because of this interface, the high level code was able to be very readable and could be easily understood without prior knowledge of the low level. This also had the added advantage of making each test run easily customizable, as all the low-level quantities could be easily and accurately changed from within the main function. Figure III shows how this interoperability worked to create a simulated RAID configuration.

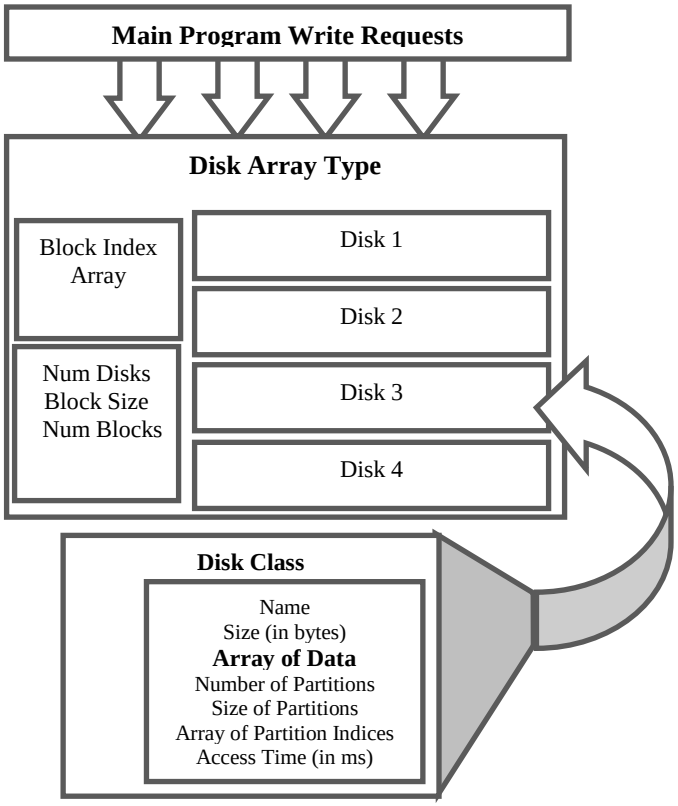

Figure III: Class level flowchart

III. Multithreading

In order to test the simultaneous write capabilities of a RAID level 0 configuration, some form of concurrency mechanism was required. For this test, the Portable Operating System Interface (POSIX) Threads Extension (commonly referred to as pthreads was implemented) [3]. POSIX threads set forth a method for providing functional concurrency in the C/C++ programming language and are defined in IEEE Standard 1003.0c [4].

A. POSIX Thread Setup

In order to account for the maximum possible concurrency of a RAID configuration of *n* disks, a set of *n* POSIX threads was created to perform the write operations. Each thread was able to perform its pre-programmed number of write operations and then terminate

B. Mutual Exclusion

It was necessary to assure that no two disks are written to at the same time. In addition to violating the physical capabilities of the read head being simulated, there also exists the possibility of having a partial data write interrupted. That is, a thread attempts to write to a location while that location is already being written by a second thread. This could potentially create corrupt data where some of the bits were written by the first thread and some by the second head resulting in a data byte that matches neither of the thread's intended write values.

Within the main program, a series of mutual exclusion flags (or mutexs) were established through the use of the POSIX threads library. These mutexs forced a writing thread to wait until the previous thread finished writing to the target disk before it can begin operation.

IV. Test Format

The main test format consisted of a random series of write operations to the established RAID configuration. For each test run, the specified number of disks were created and added to the disk array data structure. The disk array was then initialized to be in a RAID 0 configuration. A series of POSIX threads, one for each disk, was also initialized. After all preliminary steps had been completed, the threads began execution.

A. Thread Execution

Once all threads were up and running, the last thread signaled all others to begin write operations and a timer was started. Each thread first generated a random block number and determined which disk contained that block. The thread then attempted to acquire the mutex for that disk. Once the mutex had had been acquired and locked (signaling that no other threads were writing to the target disk) the block was written and the mutex was subsequently released. Each thread repeated this process for a set number of iterations and then terminated.

It should be noted that the actual data being written didn't matter so long as the write action was completed. For the purposes of this test, each disks was initialized to be filled with *0x00* bytes and each write operation filled the block with *0xFF* bytes. Also while each operation within this test was a write operation, read operations could also be simulated by some simple alterations to the program structure. In reality there is a slight difference between read and write times for hard drives; however, this difference is insignificant for the purposes of this simulation so long as each access is consistent.

B. Configurable Parameters

For the sake of consistency, each drive used within the test was initialized to be exactly identical. Each drive was kept to 512 bytes, with 32 byte partitions. While these parameters are astronomically smaller than typical consumer hard drives, they maintain all the same properties while providing for much briefer test runs. As previously mentioned, the disk access time for each block was configurable and set to a 14 ms delay. Each thread wrote 20 blocks during its execution span. This leads to a total of (20 * *n* * 32) byte accesses per test run, where n is the number of disks in the RAID configuration.

In order to obtain accurate results, each test was run 20 times and used to calculate an average execution time. These 20 runs were repeated for RAID configuration ranging in size from 2 disks to 13 disks. Performance improvements began to become less significant when larger number of disks were included, in addition to having extremely long run times.

Therefore results were limited to 13-disk arrays for this experiment.

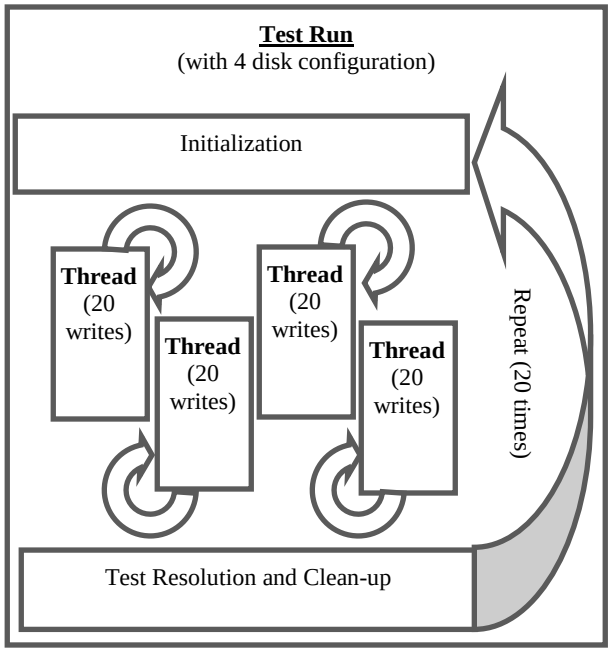

Figure IV: A flow diagram of the series of test runs for a 4-disk RAID configuration

V. Algorithmic Complexity

As previously discussed RAID 0 configuration provides substantial increases in hard drive performance. With an array of *n* drives, it is possibly to be writing data to all drives simultaneously, therefore providing a read/write rate *n* times faster than that of a single disk. That is, read/write time $t_{r/w}$ is inversely proportional to the number of disks *n*. This is demonstrated in equation 1.

$$t_{r/w} = \frac{1}{n} \tag{1}$$

However, this assumes the best case situation, where read/write requests are all sequential and all disks are always being written to. In reality, disk accesses are more random and there will frequently be higher demand for one disk causing accesses to pile up for one disk while others sit idle. Therefore, the read/write time is more accurately demonstrated in Equation 2 where *k* is some constant that decreases the slope of the exponential curve. Such is the average case complexity of RAID 0 configurations.

$$t_{r/w} = \frac{1}{n^{\frac{1}{k}}} \tag{2}$$

Put in other terms, it can be said that the read/write time is a function of *n.* Equation 3 shows this equation put into Big-Omega notation. Equation 3 represents the lower bound of the function and therefore, its best case algorithmic complexity.

$$f(n) = \Omega\left(\frac{1}{n}\right) \tag{3}$$

VI. Results

The final runtimes were achieved by calculating the time difference from when the POSIX threads began execution and completed execution for each test run. These times were averaged over the 20 test runs for each RAID configuration. Because tests with more disks involved more bytes being written, the runtime was divided by the total number of bytes written according to equation 4 in order to obtain a write time per byte. These write times were calculated for test run of each array size and output to a spreadsheet in *.csv* format. Using the analysis functions of Microsoft Excel, this data was averaged for each test run. The resulting average was then analyzed and plotted.

A. Average Case Complexity

The experimental average case complexity of a RAID configuration, containing from two to thirteen drives, is displayed in Figure V. The average case represent random block write requests from each thread. The equation of the resulting trend line is further displayed in Equation 4. Equation 5 has been formatted to match the style of Equation 2 for comparison.

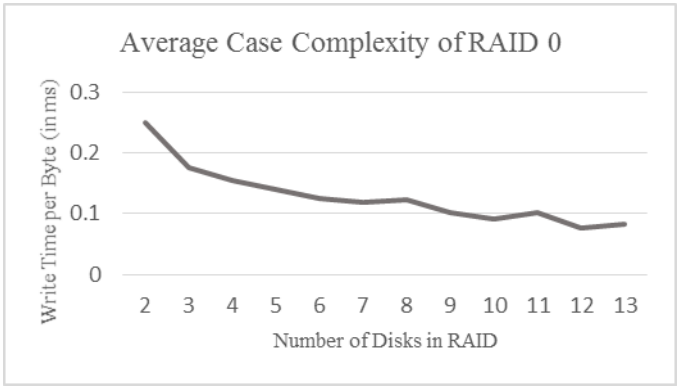

Figure V: Graph of the experimental results of RAID 0 average case write simulation

$$f(n) = 0.2505\, n^{-0.434} \tag{4}$$

$$f(n) = 0.2505\, \frac{1}{n^{0.434}} \tag{5}$$

B. Best Case Complexity

The average case complexity of a RAID 0 configuration was also determined. This graph is displayed in Figure VI. The perfect case represents a situation where all disks are continuously being written to achieve maximum throughput. This was achieved in the simulation by requiring each thread to write continuously to only one disk. The equation of the resulting trend line is further displayed in Equation 6. Once again, Equation 7 has been formatted to match the style of Equations 2 and 5 for comparison.

For the sake of completeness, a separate trial was run in which a number of write operations were run consecutively on only one disk to obtain a reading for a single drive (represented by the 1 marker on the x axis in Figure VI). This would have been impossible in the average case complexity as there is no random distribution possible with only one disk.

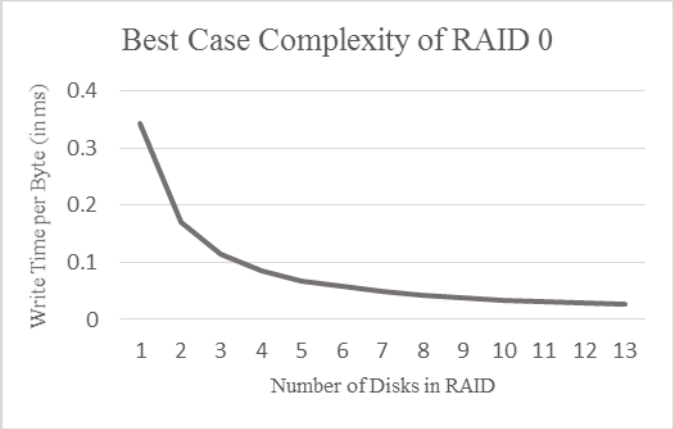

Figure VI: Graph of the experimental results of RAID 0 best case write simulation

$$f(n) = 0.3412\, n^{-0.998} \tag{6}$$

$$f(n) = 0.3412\, \frac{1}{n^{0.998}} \tag{7}$$

VII. Analysis

As shown above, it is clear that the best case complexity is nearly identical to the inversely proportional relationship proposed in Equation 1. The slight difference in the exponential can easily be accounted for by the overhead calculations inherent in calculating disk partition addresses. Furthermore, the trend line follows the experimental data precisely with no deviation. This confirms the inversely proportional ratio that is theoretically possible with RAID level 0 configuration.

However, this is often not the case in reality, as Equation V shows. There is a significant constant factor minimizing the slope of this equation. This is clearly evident when comparing the graphs as the exponential slope of Figure VI is much greater than that of Figure V. The random distribution of data in the average case test causes a much lesser slope that rarely approaches that of the best case scenario. The calculated average in Figure V and the associated trend line demonstrate a constant factor of approximately .434 that degrades the performance of realistic RAID 0 setups.

VIII. Conclusion

RAID 0 configurations can provide drastic increases in hard drive read/write performance. This is evident in the nature of the disk setup and the experimental results obtained here confirm this. While the ideal use case of a RAID 0 configuration provides a read/write time increase of *1/n*, reality often exhibits less significant increase due to the random nature of disk accesses. This will vary for each test run and a scheme that emphasizes spatial locality can greatly enhance this performance increase. In the end, RAID 0 still provides a reliable method of increasing the performance of large data storage setups.

References

[1] D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," University of California, Berkeley, CA, 1988.

[2] "Tom's Hardware Guide," Purch Company, 2014. [Online]. Available: http://www.tomshardware.com/charts/2014-mobile-hdd-charts/-08-Write-Access-Time-h2benchw-3.16,2986.html. [Accessed 10 December 2015].

[3] "The Single UNIX Specification, Version 2," The Open Group, 1997. [Online]. Available: http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html. [Accessed 10 December 2015].

[4] *IEEE Standard for Information Technology--Portable Operating System Interface (POSIX®) - System Application Program Interface (API)*, IEEE Standard 1003.1c, 1995.