Redundancy and Randomization as Effective Tools for Improving Performance

Yitzhak Birk
Viterbi Faculty of Electrical Engr.
Technion – Israel Inst. of Technology
Haifa, Israel
birk@ee.technion.ac.il

Abstract— **Redundancy is widely used for fault tolerance: storing multiple copies of data, executing the same instructions on multiple processors, providing alternate paths in a communication network, etc. However, redundancy can also be used for improving performance and even cost-performance. In some cases, combining it with randomization is highly beneficial: randomization makes performance insensitive to specific scenarios (e.g., balances average load), and redundancy is used to handle the rare "bad" cases. Moreover, this approach often results in the ideal combination of "selfish" policies with "good citizenship". We illustrate this using examples from our own past research, and offer important insights. The ideas are applicable to numerous fields.**

Keywords— Redundancy; randomization; performance; system optimization.

I. Introduction

A. Redundancy

Redundancy is the expenditure of extra resources. This can be "static", e.g., keeping multiple copies of data or constructing a network topology that provides multiple paths between any source and destination; or, it can be "dynamic", e.g., actually sending two copies of a message over two alternate paths in order to increase the probability that at least one of them reaches the destination. We will refer to them as the actual redundancy and as the exploitation of the redundancy, respectively.

B. The cost of Redundancy

Redundancy comes at a cost, and we will similarly distinguish between the cost of the redundancy itself and that of exploiting it.

Consider, for example, storing two copies of data on disk. The cost of the redundancy itself is 2X in storage space. The write cost is also 2X, as we must write two copies. The read cost, however, is zero, because we can read either copy.

C. Exploiting redundancy for performance enhancement

Redundancy is most commonly used for fault tolerance. However, it can also be used to improve performance; let us return to the two-copy example to illustrate this.

Take a closer look at reading in this seemingly trivial example. For simplicity, assume that storage capacity was doubled by doubling disk capacity without changing the number of disk drives. We said that there is no cost to reading; in fact, there is actually a benefit: given a request to read a block, the controller directs the request to the disk with the shorter response time, be it because of a shorter queue or, even if there is no queue, to the one whose current head location and rotational position are such that the data will be reached in less time. Clearly, this brings a response-time benefit. However, if the choice is based on the disk's state when it begins to serve the request, there is an additional benefit: the amount of work (disk time, like person hours) consumed in order to serve the request is reduced. This reduction in the amount of work per task translates directly into higher maximum throughput; for any given throughput, it translates into shorter queues and thus a further reduction in response time.

This simple example illustrates another, unfortunately rare in the systems world, benefit: selfish behavior (submitting the request to the disk that can respond in the shortest time) also helps balance the load on the system and increase its throughput and response time to other requests. In other words, greedy optimization and good citizenship at the same time.

D. Randomization in conjuction with redundancy

Randomization is often used in order to decouple performance from specific use scenarios, to balance the load on a set of resources, etc. However, with the benefits of "random" come the drawbacks: occasionally, hot spots may develop. As will be shown later, randomization and redundancy can work together: randomization makes things good on average, and redundancy is exploited to take care of the outliers.

E. Organization of this paper

The remainder of this paper will provide a diverse set of examples, taken from the author's own past work in a variety of application domains. Some of them will provide additional insights that will be highlighted, and together they will give the reader a toolbox of ideas and approaches, some of which are likely to be useful in the reader's future work. For brevity and in order to help build the big picture, only highlights and examples will be provided, omitting detailed derivations, optimizations and algorithms.

978-1-5386-6378-3/18/$31.00 ©2018 IEEE

II. Increasing Deadline-Constrained Throughput of Satellite-Based Transaction Processing Networks [1]

Consider remotely located stores that communicate with a point-of-sale transaction processing hub via a network that goes through a geosynchronous satellite. The uplink uses a set of contention channels, and the downlink uses a separate channel over which only the hub transmits. The messages are very short relative to the propagation delay, so channel sensing is useless. Instead, multi-channel slotted ALOHA is employed. A remote terminal picks a channel (among 100 or so) randomly and transmits. If the hub receives the transmission, it sends an Ack. In the absence of an Ack, the terminal understands that there was a collision, picks a channel randomly and retransmits. This repeats until success.

Let us begin by fleshing out the meaning of "performance" in this setting. The service provider pays a flat fee for the satellite channels yet is rewarded per transaction, so his interest is maximizing throughput; the cashier obviously wants minimum delay, and the two are conflicting desires. However, a deeper examination of the process reveals that the cashier actually has to carry out certain operations, such a handing back the credit card to the customer, before turning to get the printout. Therefore, delay minimization can be replaced with a deadline constraint. Finally, since this is a probabilistic scheme, there are no guarantees; however, since there are other possible causes for long delays (e.g., dropping the card, wrong card, etc.), it suffices to meet the deadline with a sufficiently high probability that makes failure to meet it due to communication negligible relative to the other reasons.

So far, we have redundancy (many channels) and randomization (choice of channel), but how can we combine the two in order to maximize performance? For simplicity of exposition, consider the following example: the deadline and hub-response time permit a terminal to transmit, realize that there is no Ack, and transmit one more time. Also, the permissible probability of failure to succeed by the deadline is 0.001.

One extreme approach is to not use the redundancy: with that, the maximum permissible load on the network, which in turn determines the maximum throughput, is such that the probability of collision of a packet is $\sqrt{0.001}$=0.03.

Again for facility of exposition, let us now further assume that we wish to operate at a higher collision probability, 0.1, which would enable higher throughput. Now, the use of redundancy is required in order to satisfy the deadline constraint.

One approach is to select some three channels and transmit three copies. The probability that they all fail is approximately 0.1^3=0.001, as required; this success is moreover achieved in the first attempt, thereby also minimizing delay. However, this comes at a cost: three channel slots are consumed.

Instead, we exploit the redundancy in a smarter way: in the first of the two permissible attempts, we only transmit a single packet. If and only if it collides, in the second and final pre-deadline attempt we transmit two additional copies. The probability of failing to meet the deadline is that of all three copies failing, just as before; however, now the mean number of copies per transaction drops from 3 to 1.2, enabling a 2.5X increase in throughput for the same probability of collision. In fact, one can even do much better for certain parameter values, by also optimizing the working point.

This idea is very broadly applicable in numerous areas and applications. In the defense application domain, for example, it is referred to by some as "shoot-look-shoot".

III. Handling Soft Failures in Complex Processors [2]

Perfectly functional processors nevertheless suffer occasional "soft" (transient) errors. These may be due to noise, being hit by particles, etc. The problem is becoming more severe as one shrinks the building blocks. For memory and busses, error correcting codes are used successfully. For the logic elements, however, the problem is more complex. A common approach is to carry out every operation twice, either on the same or on different hardware, and compare the results. If they are different, the computation is repeated a third time and the majority wins. (Recall that the faults are transient and "random".)

It has been observed that in a complex processor the ALUs are highly underutilized. In view of that, it was suggested in [2] to execute twice on the same processor, assigning low priority to the second computation. Computing proceeds based on the result of the first executions, but is marked as speculative until matching results of second executions are received. In the (rare) event of a mismatch, the situation is handled. It was estimated that speed and power would suffer by several percent.

A closer look at the findings revealed that the resulting rate of undetected transient failures is much lower than the permitted maximum, namely an overkill. So, it was proposed to slightly increase the clock frequency (gaining back the lost throughput and perhaps even more) and to slightly lower the operating voltage (gaining back the power efficiency). Both of these actions are likely to increase the rate at which soft failures occur, but the undetected error rate can still be well below the permitted maximum. The potential result is that the judicious use of redundancy will achieve both the originally-intended reduction in undetected soft failure rate and a performance increase with no power penalty. (This was offered as a conjecture.)

IV. Plugging the Multi-Get Hole [3]

Consider a front end server and a back end server. The former receives requests from users, sends them to the latter, receives a response and relays it to the user. An example is the Facebook system, and a request may be to show the records of John's friends.

As the number of users and/or level of activity rise, the back end server cannot meet the demand. So, it was proposed to partition the data among two such servers. This was done, to no avail. It turned out that most of the work of the back end server is per request, regardless of the amount of requested data (within reason). Since, in all likelihood, some of John's friends are on each of the backend servers, a request for those friends went to both of them, so the request rate seen by each did not decrease. (2/2 = 1/1…).

In [3], it was proposed to mitigate this processor bottleneck problem by adding memory rather than processors, and duplicating records "randomly". Now, the front end server that receives a request (and knows the mappings) derives a minimum-cardinality subset of back end servers that jointly hold the records of all (or perhaps enough) of John's friends. This is an approximation of minimum set cover.

In an actual system, there would be a master copy that is always present, and additional copies would be managed locally by each back-end server (caching). Consequently, the actual number of copies of each record is not fixed, and the amount of extra memory is a cost-performance trade-off.

So, by using redundancy (extra copies of the records) and randomization (the "random" placement of copies rather than having pairs of back end servers that mirror each other), performance was increased.

V. The RandomRAID video server [4]

A video server is a communication-intensive storage system. Its storage capacity is obviously bounded from below by the amount of data that it must store. However, the number of drives (and thus cost) must be such that it can generate the required number of concurrent video streams. Moreover, the nature of applications like video-on-demand (VOD) is such that the user expects a very high quality of service, namely virtually no "glitches" due to temporary starvation for data.

The transfer rate of even a single magnetic hard drive is hundreds of Mbits/sec, 10X-100X the rate of a compressed high definition video stream. Therefore, all video servers read data in chunks of contiguous data into buffers, out of which they stream the data over the network. The challenge is thus to maximize the (nearly) guaranteed number of concurrent video stream for given resources or to minimize the resources (the number of drives, RAM buffer size, and total storage capacity) required in order to achieve the required streaming capacity.

The effective data rate of a hard drive depends on the size of chunks of contiguous data that are read from it, asymptotically approaching its transfer rate. Therefore, a typical chunk size is such that the disks are used efficiently, striking a trade-off between RAM buffer size (increases linearly with chunk size) and the disk throughput (which affects the required number of disks). In the remainder of the discussion, we will assume that a chunk size has been chosen, and will not discuss it further.

The demand pattern (who wants to view what when) is highly unpredictable, so virtually all video servers stripe each movie across all disks, thereby guaranteeing a balanced communication load across the disks. The common wisdom has moreover been to place chunks of any given movie on disks in a round robin fashion. Also, disks are arranged in groups whose size equals that of the parity group size chosen for fault tolerance. (For example, if using a $k+1$ scheme, the disks are arranged in groups of $k+1$.) With this approach, reading data for any given video stream entails accessing the disks in that order, and joint time-space scheduling of all active streams is simple and convenient. Unfortunately, this does not work well in practice: a stream's data rate may vary; the transfer rate of a disk drive varies with radial location of the data (outer tracks contain more data than inner ones, and the angular velocity is fixed, so the transfer rate is higher when reading from outer tracks); a disk may have occasional read errors or take time out for calibration; etc. Worse yet, when a disk fails, the load of covering up for it is shared only among those in its groups. Since movies are striped across all drives, the crippled disk group is the weakest link in a chain, resulting in significant degradation in the server's total streaming capacity. The situation is even worse when a new disk is inserted, and the data of the failed disk has to be reconstructed onto it by reading from the other disks of the group. These problems become critical when the load is high and there is no slack, and require one to either refrain from operating at high load (i.e., reduce the streaming capacity) or add significant RAM buffer space to mask the resulting delays.

Instead, the Random RAID architecture takes a very different approach: when recording a movie, k consecutive data chunks along with the parity chunk constructed from them are placed in $k+1$ "randomly chosen" disks. When playing a stream, it appears as if the disks are chosen randomly, so the load is balanced on average regardless of user activity patterns.

To any given disk drive, it appears as if each chunk-read request generated at the server tosses an (unfair) coin: with N disks, it chooses the given disk with probability $1/N$. The behavior of the queue length is well understood, and when the load is very high the queue can be long with non-negligible probability, requiring advance buffering of data in memory in order to prevent visible glitches due to temporary starvation. However, here comes redundancy to the rescue.

Consider a group of $k+1$ consecutive same-stream chunks that form a parity group. The controller examines the queue lengths to the disks holding those chunks. If the length of a queue to a disk holding a data chunk is longer than some threshold value and that of the queue to the disk holding the parity is shorter by at least some margin, the controller accesses the parity chunk and the other data chunks, and refrains from accessing the long queue. Here, as was the case in the simpler mirroring example, doing this both reduces response time and helps balance the queue lengths.

Additional benefits are: the number of disk drives can be increased incrementally, not only in increments of $k+1$; when a disk fails, the parity groups of the chunks that is stored cover all other disks in the system, so the load of filling in for that disk is spread among all of them. Since the total amount of work remains unchanged, the performance degradation is minimal. Similarly when rebuilding the disk.

For the sake of brevity, we omit a detailed discussion, choice of parameter values, simulation results, etc. Instead, we now discuss the cost of exploiting the redundancy, broadening the discussion to a Random RAID used for various applications, and show that this is highly application dependent.

A. Single-block read

Normally, this entails the reading of a single block from the relevant disk. Refraining from accessing that disk brings about the need to access all the other disks holding blocks of that parity group. So, if a parity group is of size $K+1$ (K data blocks and a single parity block), there is a K-fold increase in the number of required block reads. This is clearly unacceptable unless this is an extremely rare event, which is not the case when using the redundancy for load balancing.

B. *Bulk data transfer*

Here, the chunks forming a parity group all belong to the same bulk (e.g., file), and all need to be read as soon as possible. Therefore, there is no penalty in disk traffic or buffering, and the computation penalty (XOR) is negligible.

C. *Streaming*

This is an intermediate situation: all chunks forming the parity group are required, but not immediately. Therefore, the decision to read a parity chunk instead of a data chunk results in the premature reading of some of the parity group's data chunks; once that is done, one can either buffer them until they are required or dump them after computing the required data chunk and read them again.

An interesting special case is when $K=1$, namely random mirroring. Here, exploiting the redundancy merely means reading the other copy, so there is no cost. (The benefit of choice for load balancing and latency reduction is the highest among RAIDs, as is the resulting throughput increase.) In fact, random mirroring is the basic technique underlying the highly successful storage architecture developed by XIV (subsequently acquired by IBM) for commercial transaction processing, with excellent load balancing being the main selling point.

When choosing to buffer, the choice of thresholds for accessing the parity chunk can be tuned to optimize the cost-performance tradeoffs.

An interesting insight gained while making this optimization is the distinction between starvation-prevention buffers and overflow (due to premature arrival of chunks that need to be stored until the time of their streaming) buffers. The former must be allocated in advance per stream and filled with data, whereas the latter can be shared and allocated on demand. In a streaming application, the need is known well in advance (except for the command to start a stream or resume it), so one can generate the chunk-read requests in advance. So doing helps prevent starvation and reduces the required starvation-prevention buffer size, but increases the overflow-prevention buffer size. Optimization yields the minimum total buffer size.

In the next section, we adapt many of the elements of Random RAID to a communication network.

VI. Prioritized Dispersal [5]

Consider a communication network featuring multiple paths from a source to a destination (topological redundancy). In addition to fault tolerance, this can also be used in order to reduce latency. One approach is to send copies of a message over multiple paths, and once the first arrives the mission is accomplished. However, this is costly because we always use the redundancy, thereby substantially increasing the load on the network, which in turn increases queue lengths and even causes packet drops, thus possibly increasing latency in most cases.

In order to benefit from the redundancy while limiting the negative effects, we mark one copy as high priority and the others as low priority. Whenever a low-priority packet competes with a high-priority one (of a different connection) for resources such as links and buffers, it loses. Consequently, the "original" packets do not suffer from the additional ones, so latency is improved or at least not harmed.

Similarly to the use of Random RAID instead of random mirroring in a storage system, here too it is possible to partition a message into chunks, compute additional (redundant) ones using a systematic erasure correcting code, and send all over different paths, with the extra chunks marked as low priority.

VII. Conclusions

Although redundancy was originally intended for fault tolerance, it can often be used to significantly, at times even dramatically, increase performance. One particularly interesting technique within this general approach is to combine redundancy with randomization: randomization takes care of the average and breaks correlation between usage patterns and performance, and the judicious use of redundancy helps "clip" the probability distributions, namely handle the rare bad situation as if they were failures.

Redundancy and its exploitation come at a cost, but this cost may be moderate, especially if the redundancy is already paid for in order to achieve fault tolerance. In fact, as was shown in the example of reading from mirrored storage, the cost of using redundancy may even be "negative", namely a benefit rather than a cost. We note in passing the importance of the "amount of work per mission" as a measure of system efficiency and the cost or benefit of any given approach.

The above list of examples and techniques is by no means exhaustive. For example, storage- and communication capacity are often maximized by "shrinking" the bits (space and time, respectively), thereby increasing the probability of bit errors but overcoming them using error correcting codes. The benefit from shrinking outweighs the loss due to the lower-than-1 code rate.

The presentation of the various techniques in this paper was sketchy, highlighting the techniques and ways of thinking. The most important message is that the approach is broadly applicable, and a technique used in one domain can often be adapted quite easily to a different one. Readers are encouraged to consider this in their own undertakings.

References

[1] Y. Birk and Y. Keren, “Judicious use of redundant transmissions in multi-channel ALOHA networks with deadlines”, IEEE J. Sel. Areas in Commun, vol. 17(2), pp. 257--269,Feb. 1999.

[2] A. Timor, A. Mendelson, Y. Birk and N. Suri, “Using Under-Utilized CPU Resources to Enhance its Reliability,” IEEE Trans. Dependable and Secure Computing, vol. 5(4), Oct.-Dec. 2008.

[3] S. Raindel and Y. Birk, “Replicate and Bundle (RnB) -- A Mechanism for Relieving Bottlenecks in Data Centers,” IEEE 27th Intl. Par. and Distr. Proc. Symp. (IPDPS), Cambridge MA, May 2013.

[4] Y. Birk, “Random RAIDs with selective exploitation of redundancy for high performance video servers”, Proc. NOSSDAV '97, St. Louis, Missouri, pp. 13--23, May 1997.

[5] Y. Birk and N. Bloch, “Improving network performance with Prioritized Dispersal”, Proc. IEEE Infocom 2000, Tel Aviv, Mar. 2000.