4. Performance Metrics and Perspectives

Performance metrics of computing systems are needed in order to evaluate the goodness of a particular solution (architecture, protocol, resource allocation/management strategy, etc.) or compare that solution to other candidate solutions.

4.1 Whose Performance?

Depending on the objectives of the performance evaluation/comparison, performance metrics can be divided into two main classes: System centric and Process centric.

- **System-centric** (or resource-centric) performance metrics evaluate how well the overall system or any subset of its resources are being used, regardless of the quality of service (QoS) delivered to the individual entities (e.g., processes, transactions, database queries) making use of that system.
- **Process-centric** performance metrics evaluate how well a particular resource consumer (i.e., a specific process, a thread, a network connection, etc.) is being served, regardless of the overall efficiency of the system resources usage.

One can understand the difference between the above two "types" of metrics by referring to our conception of "processes as resource consumers". Basically, one can look at "how well are the resources consumed" and/or we can also look at "how well served are the entities consuming the resources". System-centric and process-centric performance metrics do not necessarily go hand in hand. In particular, a solution that results in "poor" system-centric performance could well be providing "excellent" process-centric performance.

System-centric and process-centric performance metrics do not necessarily go against each other either. Thus, it is possible for both system-centric and process-centric performance metrics to be

quite "good."

How we quantify such goodness (i.e., "excellent," "poor," "acceptable," etc.) is the subject of this part of the course.

A Motivating Analogy: To elucidate these concepts, we use an analogy (by now, you probably realize that I like to use analogies quite a bit!) Consider a restaurant with a maximum seating capacity of 50. From the perspective of the restaurant owner, the goal is to make the most money, and thus to make sure that there is no shortage of people waiting to be seated, and that every burner in the kitchen is being utilized all the time. This will imply that food is being sold and profit is being made at the maximal rate possible (given the kitchen resources, cook's capacity, waiters' ability to serve food, etc.). From the perspective of a customer, the goal may be to eat as soon as possible (i.e. not having to wait in a long line, or not having to wait for a long time for food to be cooked, etc.) Obviously, getting the quickest possible service is unlikely when the restaurant owner insists on having people wait in line to maximize restaurant resource usage. That's why restaurants accepting reservations are usually expensive – client pays for good QoS.

Now, let us translate the above analogy to a scenario we may encounter (as a matter of fact we always encounter) when evaluating the performance of a computing system – the tradeoff between the efficient use of a resource and the quality of service delivered by that resources to its consumers. Let us look at the particular example of the service rendered by a communication link to communicating processes. Here the resource of interest is the communication link (e.g., a network link connecting two routers, or connecting a local area network (LAN) to the Internet, etc.), whereas the consumer of that resource is the connection between two parties (e.g., an HTTP server and a Web browser). From the perspective of the owner of the communication link, it is desirable for the link to be highly utilized. For instance if the link can support up to 100 Mbps, the owner would like to see it operating close to that limit, otherwise they would be better off buying (or leasing) a slower and cheaper link. From the perspective of the consumer of the link bandwidth (the HTTP connection between the browser and the server), it really does not matter if the link is efficiently utilized or not, what maters is that the connection gets the resource whenever it needs it (i.e., it does not have to "wait" for it). Clearly, if the resource is used very lightly (i.e., there are not too many other connections contending for the use of that resource), the performance delivered to the consumer (the connection) will be "excellent". From the perspective of the user accessing the web using the browser, web pages will load up in no time. However, if the resource is used quite heavily and is operating close to or at its capacity, then it is highly likely that when a consumer (i.e., the connection) wants to use it (i.e., has data to send over it), that consumer will have to "wait" for other contending consumers. As a result, the performance delivered to the consumer (the connection) will be "poor". And, from the perspective of the user accessing the web using the browser, web pages will load up sluggishly.

Achieving a good balance (tradeoff) between these two types of performance metrics is the "art" of system design.

4.2 System-Centric Performance Metrics

We model a computing system as a set of resources (or subsystems) that are being interconnected to provide a specific service in response to requests for that service.

4.2.1 Utilization

Consider a system resource S (e.g. CPU, disk, etc.) over a period of time T, as shown in Figure 4.1.

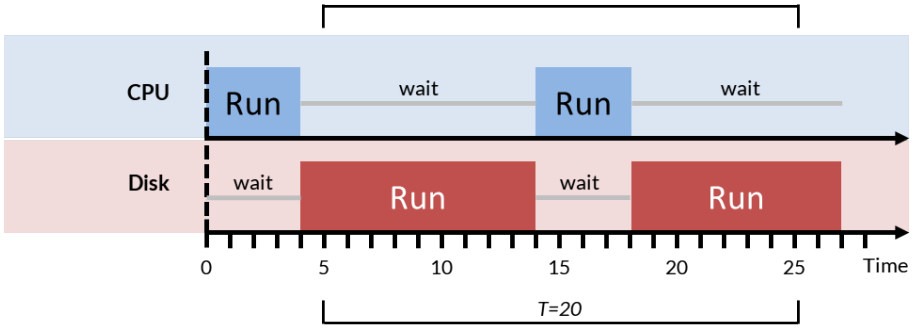

Figure 4.1: Utilization as fraction of time in an interval T during which a given resource is busy.

Assume that the resource is *idle* for I units of time. We define the **utilization**, U, of the resource S over the period T as the percentage of time that S was "in use." Thus, $U = 1 - I/T$. The maximum possible utilization of any resource in the system is therefore 1 (i.e. 100%). Looking at Figure 4.1, and taking T as depicted, we have that the utilization of the CPU is $U_{cpu} = 4/20 = 20\%$, and that $U_{disk} = 16/20 = 80\%$.

To motivate the concept of utilization, let us revisit our restaurant analogy. There are many "resources" that are needed for the restaurant operation – namely, the various stations in the kitchen, the cooks, the waiters/waitresses, the tables/seats, etc. At any given time, each one of these resources may have a different utilization. For example, while all the tables may be occupied (i.e., utilization of tables = 100%), some of the seats may be vacant, some of the cooks may be idle, and some of the kitchen stations (e.g., ovens, burners, etc.) may be idle.

It is important to note that "utilization" is a resource specific metric (i.e., it is defined for a given resource). In a computing system, we are likely to have multiple resources such as the CPU, memory, disk, Network Interface Card (NIC), etc., each of which may have a different utilization. Thus, defining (or capturing) a measure of **system utilization**[1] is elusive since it requires us to figure out the "capacity" (see below) of the entire system, which is typically limited by the capacity of the "bottleneck resource". As we will see in later parts of the course we can do this analytically as well as empirically.

[1] Recall that the notion of multi-programming was introduced to improve the "utilization" of the various system resources by allowing multiple resources to be utilized concurrently.

4.2.2 Throughput

As we made clear above, the utilization metric is useful in characterizing how individual resources are being utilized, but cannot be used to capture how well the system (as a whole) is operating. To measure this "holistic" performance of a system, we simply measure the number of "requests" it is able to "process" per unit time. This is called the *throughput* of the system.

Consider the overall system over a period of time T. Assume that the number of requests serviced by that system over the period T is N. We define the **throughput**, H, of the system over the period T as the average **rate** at which requests are being serviced. Thus $H = N/T$.

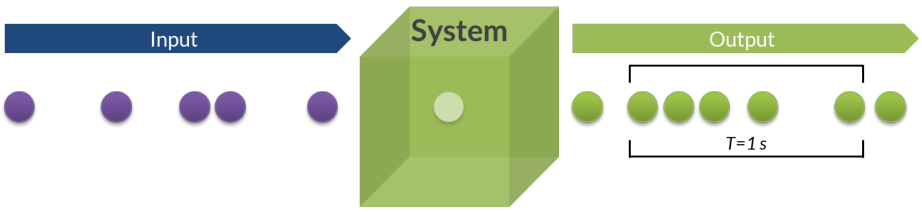

Figure 4.2: Throughput as number of requests completed over in an interval of time T.

Returning to the restaurant analogy, we can measure the throughput of the restaurant operation as the number of customers served per unit time – e.g., number of people that walk out of the restaurant with full stomachs per hour, or requests per second in case of a processing system as illustrated in Figure 4.2.

4.2.3 Capacity

The throughput of the system depends on many aspects. In particular, if there are no service requests over the period T, then the system is not expected to service any requests, thus resulting in a throughput of zero. Thus, it is important to note that "low" throughput is not necessarily indicative of a "bad" system performance. In particular, it could be indicative of a lack of requests for service from the system (e.g. throughput of restaurant at 10 am in the morning is 2 customers per hour simply because at this hour, not too many customers are interested in a meal!).

As the demand for system services[2] (e.g. measured as the rate of service requests) increases, one would expect the throughput to increase. At some point, however, the system will reach a point beyond which its throughput does not increase as a result of an increase in demand (as a matter of fact, its throughput may start decreasing as demand increases). We refer to this upper bound on throughput as the **system capacity**, as shown in Figure 4.3.

The capacity of a system is typically reached when one (or more) of the system resources becomes a **bottleneck** (i.e. its utilization approaches 100% – for real systems, a resource becomes the bottleneck at a much lower utilization percentage!).

For a real system, determining which resource will become the bottleneck that limits the

[2]The rate with which requests are made for service is also called the "offered load."

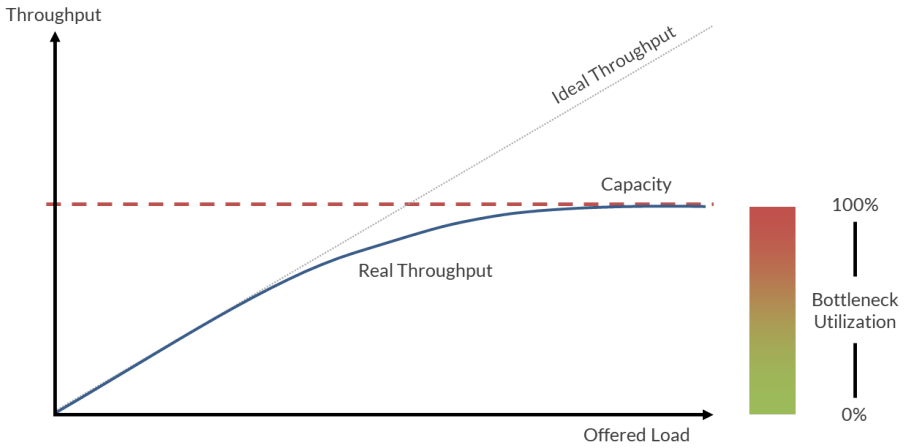

Figure 4.3: Capacity as maximum throughput achievable by the system, which occurs when the bottleneck is fully utilized.

throughput of the system is (by far) one of the most fundamental questions that a system designer must ask (and answer, through experimentation and analysis).

Knowing the culprit (when throughput reaches its maximum) is important because it points system designers to the resource that needs "upgrading" (or the process within the system that needs improvement, optimization, streamlining, etc.).

It is important to note that the "bottleneck" of the system depends on the nature of the load to which the system is subjected. For example, a given system may have the CPU as the bottleneck when most of the tasks in the system are computation intensive. The same system may have the I/O (disk or network) as the bottleneck when most of the tasks in the system are I/O intensive

4.2.4 Effective Throughput (or "GoodPut", or Bandwidth)

In many instances, the service rendered to a particular request is not of any value, and thus system resources are consumed needlessly. The following are examples.

- **Missed Deadlines**: If a request is associated with a deadline (by which service must be finished), then if that deadline is missed, the service rendered is of no value to the system (since the process requesting that service cannot benefit from it—and thus won't pay for it!). For instance, if a video frame is transmitted late, then it is of no value to the receiver (and thus to the system). One analogy here is a restaurant that does not charge its customers if food is not served by a certain deadline (or, pizza is free if delivered more than 30 minutes after being ordered, or a stock trade is free if not executed within 2 minutes, etc.)
- **Redundant Work**: Consider a system in which (due to a glitch, or a system failure) a request is serviced twice. Obviously, the extraneous "work" done by the system is of no value and is thus a waste. An instance of that is the "retransmission" of packets on an Ethernet Local Area Network (LAN) due to collision. Another instance is the abortion of transactions due to concurrency control constraints, or the processes necessary to protect against (and recover

from) failures, etc.

- **Necessary Overhead**: In many instances, system resources are consumed due to "necessary" overhead. For example, an Ethernet switch may be rated at 10 Mbps, however, the "real" bandwidth (in terms of data that applications can communicate over the wires) that one can "squeeze" out of it (even ideally) may be less because the Ethernet protocol requires all sorts of "extra" framing bits to be sent alongside the data. The same is true of IP packets or ATM cells. For Operating Systems, we encounter the same type of "necessary overhead" (e.g. CPU cycles that are wasted to implement the scheduler, the interrupt service routines, etc.)

If $N' < N$ is the total number of completed service requests "of value" over a period T, then the effective throughput, $H' = N'/T$. The terms **goodput** and **bandwidth** are often used to refer to **effective throughput**.

A Streaming Media Communication Example: An MPEG application reported that 10% of all packets received were "late." If the network throughput is 2 Mbps, and 5% of the packet size goes towards header, etc. What is the *effective throughput*?

To answer this question, we must figure out the portion of the traffic that is "good" or useful for the application at hand. Packets that arrive late or portions of packets that are needed for things like "addressing" and "routing", while absolutely necessary, could be construed as "wasted" from the perspective of the application[3]. From the above description, 10% of the 2 Mbps are wasted (because the packets are lost). That leaves us with 90% (i.e., 1.8 Mbps) of the packets. But, additionally, 5% of these packets are useless (e.g., because they are used by lower layer protocols), this leaves us with 95% of the 1.8 Mbps, which is 1.71 Mbps of "goodput" or effective throughput.

A Computer Architecture Example: A pipelined CPU design is able to process one instruction per clock cycle by dividing the execution of such an instruction into four stages (e.g., Fetch, Execute, Memory Load and Memory Store). Assuming that the clock speed is 1 GHz, one can see that the bandwidth (in Millions of Instructions Per Seconds, or MIPS) of this CPU could be as high as 1,000 MIPS. However, when conditional branch processing is involved, it is not possible to fill the pipeline since the destination address is not known until the end of the third stage. This means that for every branch instruction, the pipeline is filled with a "bubble" for two clock cycles (this is called a pipeline "hazard" and is but one of many such hazards that compilers must anticipate and deal with. Assuming that 15% of all instructions in a program are conditional branches, what would be the "effective throughput" of that CPU? One may follow steps similar to the above to answer this question.

4.3 Process-Centric Performance Metrics

Consider a set of n service requests (e.g. restaurant customers, printer jobs, HTTP requests to a web server, etc.). Assume that these requests are being submitted to the system at times $t_1, t_2, \ldots, t_n$ and are completed at times $t'_1, t'_2, \ldots, t'_n$ respectively.

[3]Here it is important to recall our emphasis on layering – an application operating at one layer, may consider the resources (e.g., bits in packet headers) used by lower layers as "wasted". Thus, what constitutes goodput is really in the eye of the beholder – i.e., dependent on what layer you are at.

4.3.1 Response Time

The **response time** R_i for a single request i is defined as the time elapsed from the time t_i when the request is submitted until the service is finished at t_i' as depicted in Figure 4.4. In other words, $R_i = t_i' - t_i$. As such, the response time for the requests in the figure will be $R_A = 11$ and $R_B = 16$.

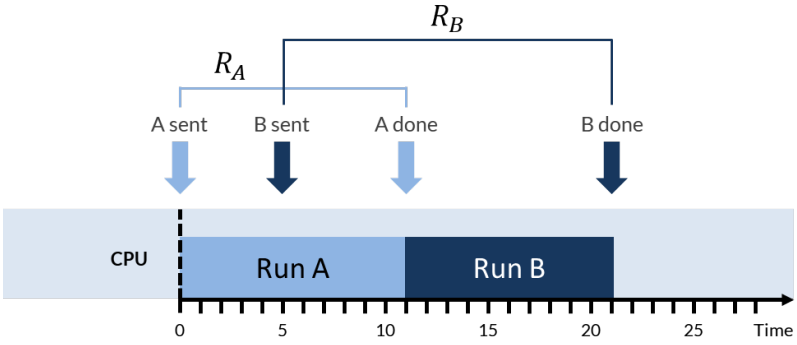

Figure 4.4: Response time for two requests A and B processed in sequence by a CPU.

If follows that the *average response time* is the mean response time measured over a large number of requests.

4.3.2 Jitter (Response Time Variability)

The average response time gives an expected value for how long it would take to service a request. The "real" response time, however, may be very different. To gain some confidence in the response time, we need to measure the variability of the response time. This variability (often referred to as "jitter") could be measured using higher moments (e.g. variance, standard deviation, etc.).

The notion of variability is important to understand. When we are told that the average waiting time at the Burger King across the street is 3 minutes, this means that the average response time was found to be 3 minutes. But, this average may describe very different response time "distributions" (a term we will understand better in the next few lectures). In particular, a "savvy" computer scientists should always try to get a feel for how much variability is to be expected around this average.

We can measure variability in a number of ways:

- **Standard Deviation (or Variance)**: The variance is the mean of the squared difference between each measurement and the average of all measurements. The standard deviation is the square root of the variance. Both of these measures give us a measure of how dispersed the data is around the mean.
- **Nth Percentile**: This gives us a measure of where the "mass" of the distribution lies. So, for example, if the median of a response time is 20 msec, then we know that half the time we would expect the response time to be below 20 msec and half the time the response time will be above 20 msec. If the 90th percentile of the response time is 40 msec, then we know that 90% of the requests in the system would be served in 40 msec or less, etc.

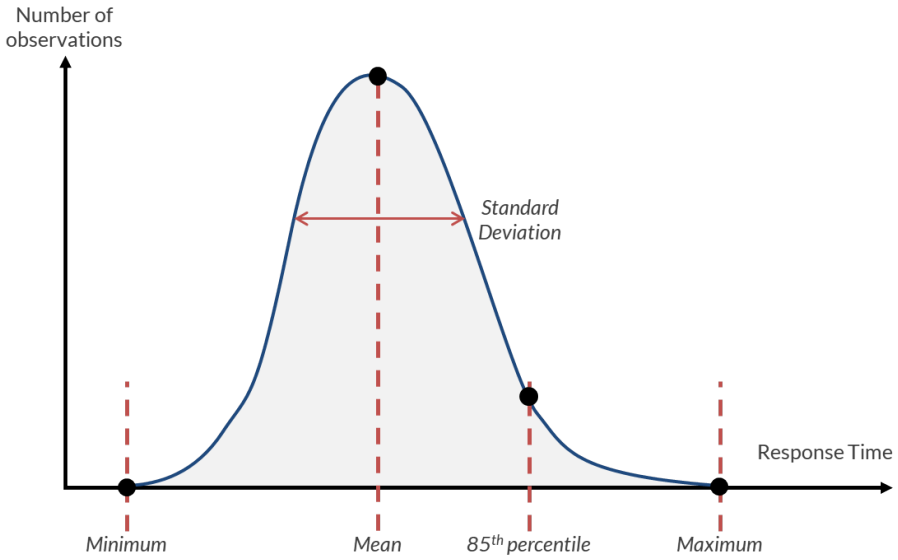

Figure 4.5: Variability of response time measured over a (large) number of observations.

4.3.3 Guarantee Ratio

Response time is not the only "measure" of goodness. Generally speaking, a process may define what constitutes acceptable "Quality of Service" (QoS). The **guarantee ratio** is defined as the percentage of service requests that are completed subject to the QoS constraints set by the processes requesting these services.

For example, if a deadline is imposed on the service requests, then the percentage of requests that end up meeting their deadlines is the *guarantee ratio*.

4.3.4 Lag and Lateness

Assume that each request for service is associated with a deadline. A request serviced before its deadline is said to have "met" its deadline. A request serviced after its deadline is said to have "missed" its deadline.

The **lag** in such a system is a measure of "how much later after the deadline service was completed," on the average. Requests that meet their deadlines have a *lag* of 0. Those that miss their deadlines have a positive lateness. The **average lag** is the mean lag over all tasks. The **average lateness** is the mean lag for late tasks.

4.3.5 Fairness

Another process-centric performance metric is related to fairness. If the performance of a system degrades (say as a result of overload), then it should degrade "equally" for all processes (assuming

all processes are of equal importance or value to the system). In particular, degradation in response time should be across all processes—as opposed to penalizing few processes while others are not. An extreme case of unfairness is worth noting. When a process is not serviced at all (i.e. the process is not allowed to make progress) while others are continuously serviced, then **starvation** is said to occur. **Fairness** (or lack thereof) is often talked about in the context of "classes" or "types" of processes (e.g. long transactions versus short transactions, CPU-bound versus I/O-bound, read-only transactions versus update transactions, user processes versus kernel processes, etc.).

To understand the notion of *fairness*, consider a set of tasks that are submitted to a system resource (e.g. CPU) for service. All tasks are identical in priority and each task requests a certain number of seconds of exclusive use of the resource. Let these requested execution times be C_1, C_2, C_3, etc. Assume that the response time for these tasks is R_1, R_2, R_3, etc. A fair system will try to keep the ratios of R_i/C_i close to each other. An unfair system will violate this property.

4.3.6 Speedup and Slowdown

Often times it is natural to compare the behavior of a system before and after a crucial change that might affect its performance. For instance, one might want to understand if a system upgrade was beneficial or detrimental—and by how much. Or perhaps one would like to understand how much more degraded is service after a surge in workload, compared to the case when the system was not under pressure.

The **speedup** and **slowdown** metrics allow to carry out such comparisons. Say that some metric of performance expressed in time was collected of a system before the change of interest—the *old* system. Call the measurement T_{old}. This could be, for instance, the average response time of user requests in the old system. Next, collect the same metric in the *new* system after the change under analysis. Thus, we collect T_{new}.

The speedup capture how many times ($\times$) faster is the new system compared to the old one under the considered metric. Therefore, we define

$$\text{Speedup} = \frac{T_{old}}{T_{new}}. \tag{4.1}$$

As you can see, if the T_{new} has improved after the considered change, the resulting speedup will be a number greater than 1. For instance, if the considered metric is response time, and the response time is half what it was in the hold system, the speedup will be $2\times$. If the performance after the change under analysis has degraded, the speedup metric will have a value less than 1.

On the other hand, the slowdown is the inverse of the speedup and returns a value greater than 1 to indicate a performance loss. It is defined as follows:

$$\text{Slowdown} = \frac{T_{new}}{T_{old}}. \tag{4.2}$$

For instance, consider a system that was previously subject to very lightweight load and for which a response time $R_{lightload}$ was observed. Consider now a surge in workload. As the amount of backlogged work grows in the queues, the response time also grows. Say that $R_{heavyload}$ is the new response time that has increased threefold. Thus, the slowdown caused by the surge in workload is $3\times$.

Box 4.3.1 Amdahl's Law

Amdahl's Law states that if the fraction of time a resource is used in serving requests is f then improving the performance of that resource x-fold (i.e., making the resource x-times faster) will improve the overall performance of the system by a factor of y (speedup), where y can be calculated as follows:

$$y = \frac{1}{1 - f \cdot \left(1 - \frac{1}{x}\right)} \tag{4.3}$$

Proof:

Let the time to complete the service of a request be T. It follows that the resource in question is used for $f \cdot T$ units of time. Improving the performance of the resource by x-fold implies that the new time it takes the "improved" resource to render its service is $\frac{f \cdot T}{x}$. Thus the total time it takes to complete the request (assuming that except for speeding up the resource under consideration, nothing else in the system has changed) would be:

$$T - f \cdot T + \frac{f \cdot T}{x} = T \cdot \left(1 - f \cdot \left(1 + \frac{1}{x}\right)\right) \tag{4.4}$$

This means that the speedup in serving requests would be:

$$y = \frac{T}{T \cdot \left(1 - f \cdot \left(1 + \frac{1}{x}\right)\right)} = \frac{1}{1 - f \cdot \left(1 - \frac{1}{x}\right)} \tag{4.5}$$

■

Notice that even if one improves the resource "infinitely," the impact of that improvement is bounded by $1/(1-f)$. If f is small (e.g. 2%), then the impact of improving the resource in question is quite limited. For $f = 0.02$ the impact is no more than $1/0.98$, which is equal to 1.02. In other words, it is not "worth it".

Qualitatively, Amdahl's law states that one should optimize the component of the system that is used "most" because that component is likely to impact the overall performance the most. Clearly, by its definition, a bottleneck in the system is the one used the "most" (since other resources would be idle most of the time).

Thus, qualitatively, Amdahl's law says that the most improvement will result from speeding up the *bottlenecked* resource. Notice that when one speeds up a bottleneck, the system may develop a different one[a]. Another way of saying this is that speeding up a system resource will change the percentage of time in which that resource (and other resources) are used – perhaps making some other resource the one used "the most."

Figure 4.6 shows the expected overall speedup when a component of the system becomes twice as fast (e.g. CPU is twice as fast). The system speedup is shown for components with varying "percentage of use." So, doubling the speed of a component used 20% of the time, will only result in a 10% improvement.

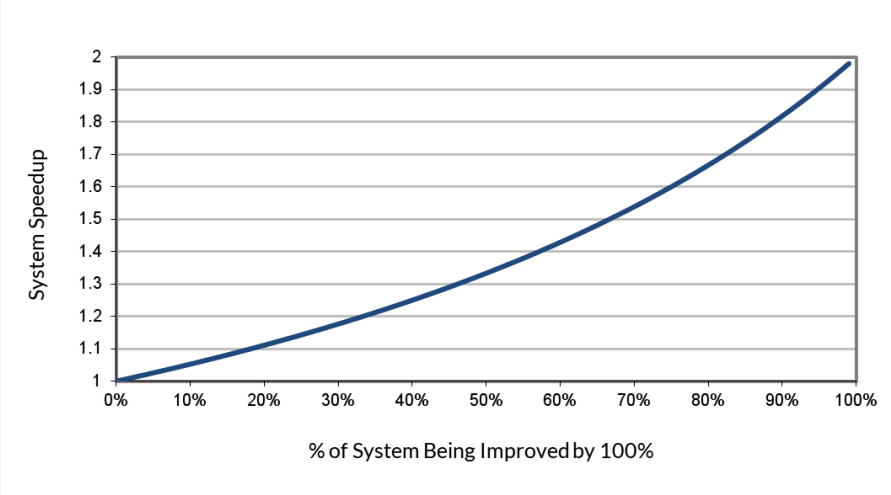

Figure 4.6: Amount of system speedup that can be achieved when the sped-up portion of the system goes from 0% to 100%.

Use Cases

An Example: The average response time of a Web server is 100 milliseconds. Of this time, 10 milliseconds are spent on the CPU, 65 milliseconds are spent on I/O and 25 milliseconds are spent on DMA operations. A friend of yours suggested that buying a new server with twice the CPU speed would cut the response time by 50%. Explain why your friend is mistaken. What would you try to upgrade about this server to reduce response time the most? Give specific numeric examples.

Another Example: To speed up memory access, caching is typically used. A memory cache is a small but fast memory where data recently accessed is kept in anticipation of future references. When an access is made, if the data is in the cache, then it is returned quickly. This is called a cache hit, otherwise main memory is accessed and the access is said to be a cache miss. Consider two possible optimizations for a memory system. The first will cut the latency of the main memory by 50%, whereas the second would cut the latency of the cache by 20%. Assuming that the latency of the main memory is five times the latency of the cache (i.e., if an access to the cache takes one unit of time, then access to main memory would take 5 units of time), which of

these optimizations would you pick if the cache hit rate is 95%? Which one would you pick if the cache hit rate is only 90%? What is the overall improvement in both cases?

In the above examples, we used words like "faster" to compare the performance of two subsystems. When one says that "performance" improves, one must (or should) be referring to a specific performance metric, with specific units. "Faster" and slower are words we would use to compare speeds, or rates, etc. but not times or delays. Similarly, "longer" and "shorter" are words we would use to compare response times or delays but not speeds or rates.

If we are talking about rates (i.e., as in bandwidth, or capacity), then saying that system X is 33% faster than system Y means that $\text{rate}(X)/\text{rate}(Y) = 1.33$.

Since we can always think about rates as the "inverse" of some time delay (e.g., clock rate = 1/Cycle-Time), then we can also translate the above to a statement about the times. For example, if one says that CPU X is 33% faster than CPU Y, then this would be equivalent to saying that $\text{rate}(X)/\text{rate}(Y) = 1.33$, which is the same as saying that $\frac{1/\text{Cycle-Time}(X)}{1/\text{Cycle-Time}(Y)} = 1.33 \implies \frac{\text{Cycle-Time}(Y)}{\text{Cycle-Time}(X)} = 1.33 \rightarrow$ Cycle-Time for CPU Y is 33% longer than that of X.

Notice that the key here is that faster/slower or longer/shorter refer to a relationship between a numerator and a denominator. So whatever ratio or percentage given to describe that a metric for system X is (faster/slower or longer/shorter) than that of system Y should be interpreted as the ratio for X versus Y for the metrics used to account for speed/rates/frequency (in case of faster/slower) or times/delays (for longer/shorter).

Parallelism and Amdahl's Law

Software and hardware go hand in hand when it comes to achieving high performance in settings where we might have concurrent processing (e.g., using parallel processing on a supercomputer with large number of CPUs or cores per CPU, or using a large cluster in a data center or in the cloud).

Programs must be written to explicitly take advantage of the underlying hardware, and existing non-parallel programs must be re-written if they are to perform well in such a setting.

A parallel program does many things at once. Just how many depends on the problem at hand. Suppose $1/N$ of the total time taken by a program is in a part that **cannot be parallelized** (sequential), and the rest $(1 - 1/N)$ is in the parallelizable part, as shown in Figure 4.8.

In theory you could apply an infinite amount of hardware to do the parallel part in zero time, but the sequential part will see no improvements. As a result, the best you can achieve is to execute the program in $1/N$ of the original time, but no faster. In fact, once the parallelizable part of a program has been distributed across enough processors, no visible amount of speedup can be further achieved, as shown in Figure 4.7^b. In parallel programming, this relationship is commonly referred to as Amdahl's Law.

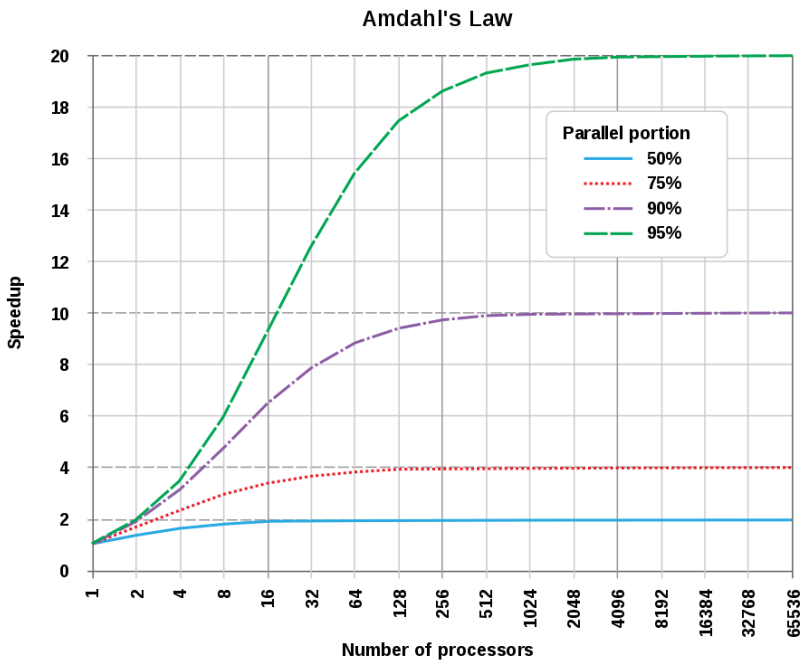

Figure 4.7: Diminishing return for speedup with increasing number of processors.

Amdahl's Law governs the speedup achieved when one part of the system is optimized (e.g., the parallelizable portion of a program) while the other part of the system is not (e.g., the sequential portion of a program). When speeding up the execution of a program through parallelization, the **speedup** S is defined as the time it takes a program to execute in serial (with one processor) divided by the time it takes to execute in parallel (with many processors):

$$S = \frac{T(1)}{T(j)} \tag{4.6}$$

Where $T(j)$ is the time it takes to execute the program when using j processors (see Figure 4.8).

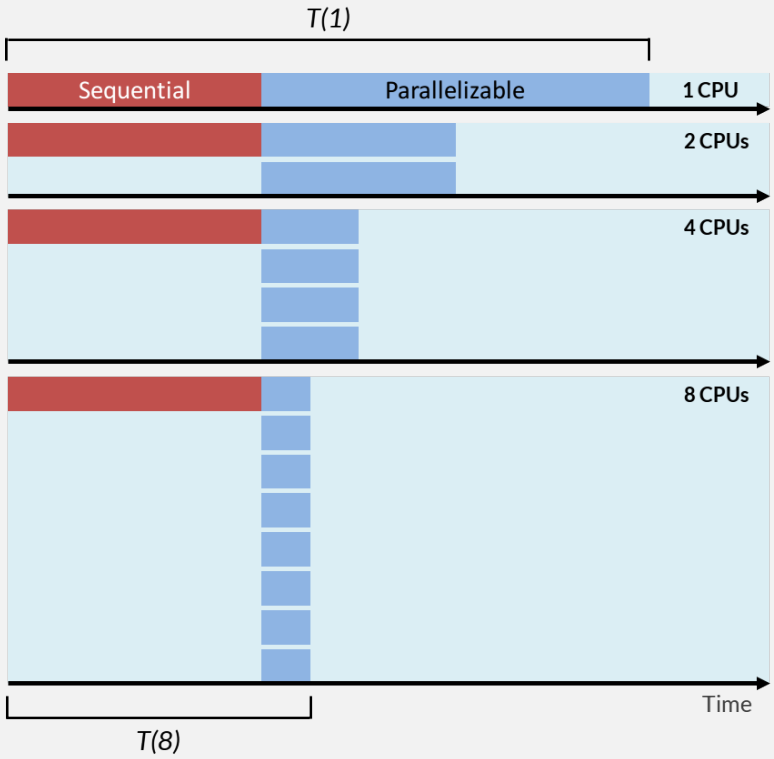

Figure 4.8: Illustration of how Amdahl's Law applies to parallelization of code.

From the above, it is clear that the real hard work in writing a parallel program is to make N as large as possible, and finding a large number of processors on which to run the parallel section of the code – we will cover this later when we discuss the use of cloud platforms for big-data processing.

By applying Amdahl's law, we can balance (or study the tradeoff) between these two alternatives (in terms of what is a better investment to speed up the overall computation).

[a] This is a common pitfall of projects aiming at alleviating highway congestion. If a section of a highway is the bottleneck that reduces speeds to 30 mph, then widening it to allow for 65 mph speeds will simply move the bottleneck somewhere else, and the highway will still move at a snail's pace (perhaps 31 mph).

[b] Figure from `https://en.wikipedia.org/wiki/Amdahl%27s_law`.

4.4 Other Aspects of "System Performance"

In this course, we will be concerned (mostly) with system performance when the system is not subject to failures. However, for "real" system, failures happen! Thus, an important characteristic of performance is the behavior of a system when failures occur.

4.4.1 Reliability and Availability

The concepts of reliability and availability are often used interchangeably (and thus confused). The **reliability** of an on-line solution is a measure of the probability that the system functions correctly (provides services) continuously *for a specified time period*. In other words, reliability is a measure of *continuity* of service. The **availability** of an on-line solution is a measure of the probability that a system is available at an arbitrary point in time. In other words, availability is a measure of readiness of service.

Both reliability and availability are "process-centric" metrics because they describe attributes of the system that are of value to the "client" of the system. The reliability of the system (or a service thereof) is a measure of how likely it is that a specific request will be successfully serviced. The availability of the system is a measure of how likely it is that the system will "accept" the request — but not necessarily complete its service of the request.

To understand the difference between reliability and availability, consider a web server that has an availability of 99% and a reliability of 97% for, say, completing transfers of less than 10 seconds in length. These figures could be interpreted as follows. If 1,000 requests are made to that web server for service, then at most 10 of these requests will experience a "connection refused" outcome — i.e. service was unavailable. Also, at most 3% of the remaining requests (requiring the web server to be continuously functional for up to 10 seconds) will result in a "connection reset" outcome — i.e. service was interrupted.

Consider Figure 4.9. Availability could be easily estimated by computing the ratio of "up time" to "up time + down time." Admittedly, there are a few slightly different definitions of availability depending on whether we are dealing with a *reparable* system, or a system that once failed cannot be repaired. Let's sticking to reparable systems, which is typically the case for computing systems that can suffer transient failures — think about a machine that buckled under heavy load and just needs a good reboot to spring back into action.

The "up time" is typically referred to as the **Mean Time Between Failures** (MTBF), which is a measure of the average amount of time between a completed repair and the next failure. If you observe a system under a long interval of time and accumulate in T_{up} the total time during which the system was up, if n_{fail} failures where observed, then MTBF can be computed as MTBF = $\frac{T_{up}}{n_{fail}}$.

The "down time" is typically referred to as the **Mean Time To Repair** (MTTR), which is a measure of the average length of time for a failed system to recover (or be repaired). Both the MTBF and MTTR are system-centric metrics — a client would not care less for these metrics! In a way that is similar to MTBF, if T_{down} is the total time that the system has spent in maintenance/under repair, then MTTR = $\frac{T_{down}}{n_{fail}}$.

With this in mind, the availability A of a system at steady-state can be computed as $A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$.

To exemplify, if the MTBF is 4 days and if the MTTR is 1 hour, then the availability of the

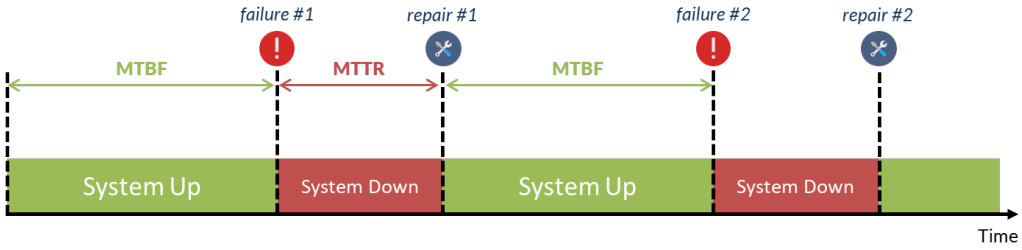

Figure 4.9: Metrics of system availability and reliability.

system would be $A = \frac{4 \cdot 24}{(4 \cdot 24+1)} = 98.97\%$. Notice also that the same availability level could result from a MTBF of 96 second and a MTTR of 1 second. The difference between the former and latter cases would be reflected in the measure of *reliability*. In particular, the reliability of the system for a period of 1 minute will be much higher in the former case than in the latter. We will explain why this is the case later in this section.

Thus, the reliability of a system is described using a function of time $R(t)$. This function is typically monotonically decreasing. The calculation of a system reliability function is slightly more complex because it involves aspects of *modeling* of the failure processes, which include aspects related to the distribution of failure events. But long story short, a widely accepted way to express reliability is by using $R(t) = e^{-t/\text{MTBF}}$. Indeed, as it will be obvious from Section 8.1.6, $R(t)$ models the probability that if the system is up (because it accepted a job/request/transaction) at a random point in time, then its next failure will at t time units or later.

As embedded in the definition above, the calculation of system reliability is parameterized by the length of time t that is required for completion of service. To understand the importance of this latter point, consider a Web banking application that supports on-line transactions (e.g. check payment, transfers, etc.). Furthermore, assume that the average transaction on the Web site was measured to take an average of 5 minutes. As exemplified earlier, assume that the MTBF for the site is 96 seconds and that the MTTR is 1 second. Under such assumptions, it is clear that the probability that there will be a 5-minute failure-free interval is almost zero, resulting in a rather unreliable system for long transactions — despite the fact that the system is almost 99% available!

4.4.2 Maintainability

Maintainability (also called **serviceability**) is another metric that is often used in conjunction with reliability and availability. Maintainability measures the average time between "service calls" and the length of time it takes to service (e.g. repair, recover, etc.) the system. Thus, under such a definition, MTBF and MTTR could be used as indications of maintainability. For software systems, however, recovery is often automated (e.g. upon crash, system reboots automatically). Thus, other metrics are used to characterize maintainability, including periodic upkeep (e.g. backups, check-pointing, upgrade, etc.).

Reliability analysis is the process of estimating (or predicting) the reliability, availability and maintainability of a computing system starting from the characteristics of its components. This

involves a determination of the dependencies between the various system components and their failure modes. From this analysis, it is possible to estimate the overall reliability of the system.

Once an assessment of the reliability/availability of a system is made (through the reliability analysis process described above), it is often the case that a higher level of reliability/availability is desired. There are several generic techniques used to improve reliability and availability, including fault avoidance techniques and fault tolerance techniques. Fault avoidance techniques aim at removing (or reducing the likelihood of) failures through a careful verification of the system components (e.g. debugging, stress testing, etc.) While such techniques are useful for most failure modes of individual system components, they tend not to be practical for larger subsystems that include multiple components integrated in a complex fashion. Also, in many instances, eliminating certain failures is extremely hard (e.g. distributed deadlocks, memory leaks) and even impossible (e.g. disk crashes). For such subsystems, fault tolerance techniques are utilized.