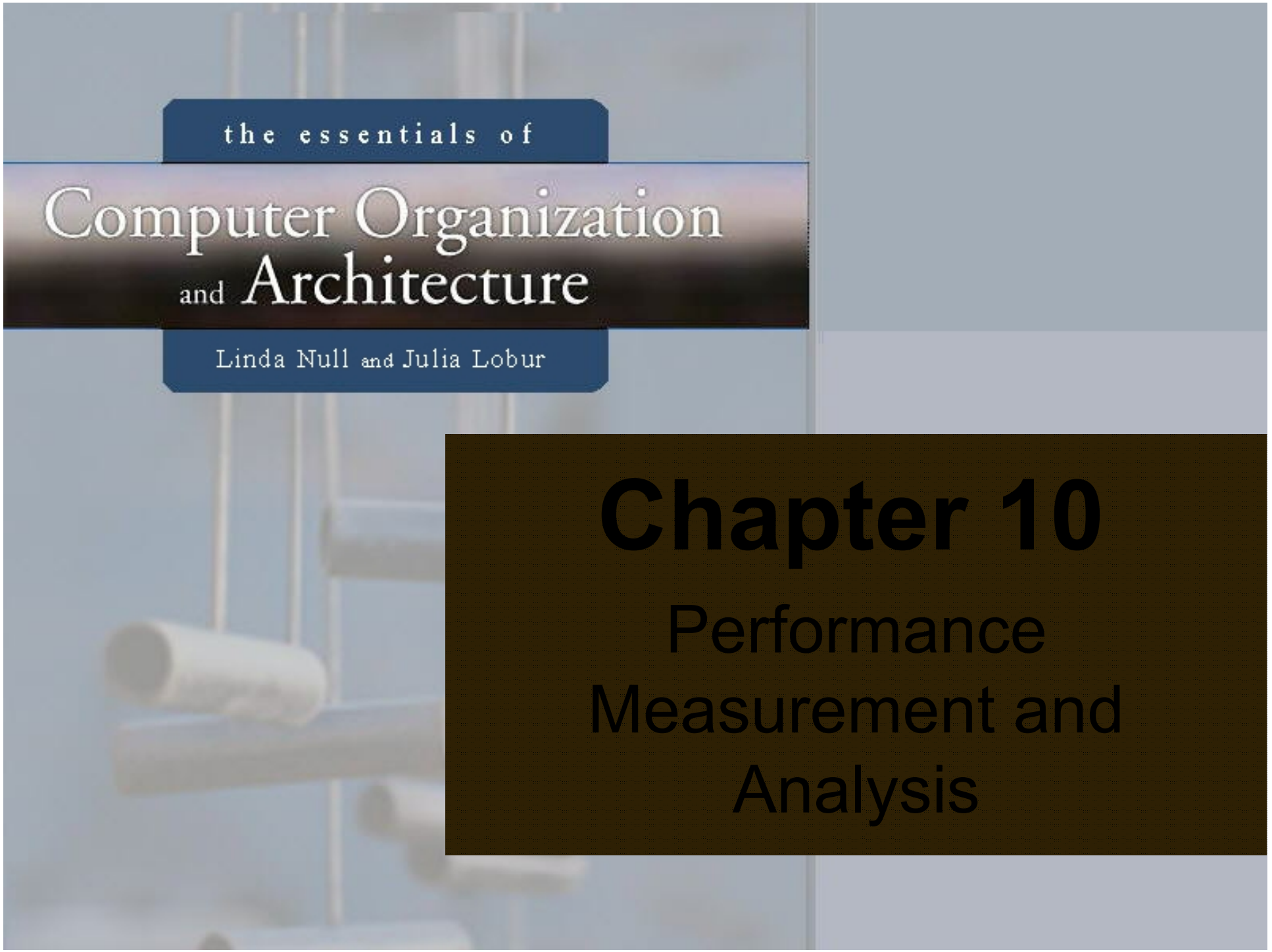


Chapter 10

Performance Measurement and Analysis

Chapter 10 Objectives

- Understand the ways in which computer performance is measured.
- Be able to describe common benchmarks and their limitations.
- Become familiar with factors that contribute to improvements in CPU and disk performance.

10.1 Introduction

- The ideas presented in this chapter will help you to understand various measurements of computer performance.
- You will be able to use these ideas when you are purchasing a large system, or trying to improve the performance of an existing system.
- We will discuss a number of factors that affect system performance, including some tips that you can use to improve the performance of programs.

10.2 The Basic Computer Performance Equation

- The basic computer performance equation has been useful in our discussions of RISC versus CISC:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- To achieve better performance, RISC machines reduce the number of cycles per instruction, and CISC machines reduce the number of instructions per program.

10.2 The Basic Computer Performance Equation

- We have also learned that CPU efficiency is not the sole factor in overall system performance. Memory and I/O performance are also important.
- Amdahl's Law tells us that the system performance gain realized from the speedup of one component depends not only on the speedup of the component itself, but also on the fraction of work done by the component:

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

10.2 The Basic Computer Performance Equation

- In short, Amdahl's Law tells us to make the common case fast.
- So if our system is *CPU bound*, we want to make the CPU faster.
- A *memory bound* system calls for improvements in memory management.
- The performance of an *I/O bound* system will improve with an upgrade to the I/O system.

Of course, fixing a performance problem in one part of the system can expose a weakness in another part of the system!

10.3 Mathematical Preliminaries

- Measures of system performance depend upon one's point of view.
  - A computer user is most often concerned with *response time*: How long does it take the system to carry out a task?
  - System administrators are usually more concerned with *throughput*: How many concurrent tasks can the system handle before response time is adversely affected?
- These two ideas are related: If a system carries out a task in k seconds, then its throughput is $1/k$ of these tasks per second.

10.3 Mathematical Preliminaries

- In comparing the performance of two systems, we measure the time that it takes for each system to do the same amount of work.
- Specifically, if System A and System B run the same program, System A is *n times as fast* as System B if:

$$\frac{\text{running time on system B}}{\text{running time on system A}} = n$$

- System A is x% faster than System B if:

$$\left[\frac{\text{running time on system B}}{\text{running time on system A}} - 1\right] \times 100\% = x\%$$

10.3 Mathematical Preliminaries

- Suppose we have two racecars that have just completed a 10 mile race. Car A finished in 3 minutes, and Car B finished in 4 minutes. Using our formulas, Car A is 1.25 times as fast as Car B, and Car A is also 25% faster than Car B:

$$\frac{\text{time for Car B to travel 10 miles}}{\text{time for Car A to travel 10 miles}} = \frac{4}{3} = 1.25$$

$$\left[\frac{\text{time for Car B to travel 10 miles}}{\text{time for Car A to travel 10 miles}} - 1\right] \times 100\%$$

$$= \left[\frac{4}{3} - 1\right] \times 100\% = 25\%$$

10.3 Mathematical Preliminaries

- When we are evaluating system performance we are most interested in its expected performance under a given workload.
- We use statistical tools that are *measures of central tendency*.
- The one with which everyone is most familiar is the *arithmetic mean* (or average), given by:

$$\frac{\sum_{i=1}^{n} x_i}{n}$$

10.3 Mathematical Preliminaries

- The arithmetic mean can be misleading if the data are skewed or scattered.
  - Consider the execution times given in the table below. The performance differences are hidden by the simple average.

Program	System A Execution Time	System B Execution Time	System C Execution Time
v	50	100	500
w	200	400	600
x	250	500	500
y	400	800	800
z	5000	4100	3500
Average	1180	1180	1180

10.3 Mathematical Preliminaries

- If execution frequencies (expected workloads) are known, a *weighted average* can be revealing.
  - The weighted average for System A is:

$$50 \times 0.5 + 200 \times 0.3 + 250 \times 0.1 + 400 \times 0.05 + 5000 \times 0.05 = 380.$$

Program	Execution Frequency	System A Execution Time	System C Execution Time
v	50%	50	500
w	30%	200	600
x	10%	250	500
y	5%	400	800
z	5%	5000	3500
Weighted Average		380 seconds	695 seconds

10.3 Mathematical Preliminaries

- However, workloads can change over time.
  - A system optimized for one workload may perform poorly when the workload changes, as illustrated below.

Program	Execution Time	Execution Frequency #1	Execution Frequency #2
v	50	50%	25%
w	200	30%	5%
x	250	10%	10%
y	400	5%	5%
z	5000	5%	55%
Weighted Average		380 seconds	2817.5 seconds

10.3 Mathematical Preliminaries

- When comparing the relative performance of two or more systems, the *geometric mean* is the preferred measure of central tendency.
  - It is the *n*th root of the product of *n* measurements.

$$G = \left[x_1 \cdot x_2 \cdot x_3 \cdot \cdot \cdot x_n \right]^{\frac{1}{n}}$$

- Unlike the arithmetic means, the geometric mean does not give us a real expectation of system performance. It serves only as a tool for comparison.

10.3 Mathematical Preliminaries

- The geometric mean is often uses *normalized* ratios between a system under test and a reference machine.
  - We have performed the calculation in the table below.

Program	System A Execution Time	Execution Time Normalized to B	System B Execution Time	Execution Time Normalized to B	System C Execution Time	Execution Time Normalized to B
v	50	2	100	1	500	0.2
w	200	2	400	1	600	0.6667
x	250	2	500	1	500	1
y	400	2	800	1	800	1
z	5000	0.82	4100	1	3500	1.1714
Geometric Mean		1.6733		1		0.6898

10.3 Mathematical Preliminaries

- When another system is used for a reference machine, we get a different set of numbers.

Program	System A Execution Time	Execution Time Normalized to C	System B Execution Time	Execution Time Normalized to C	System C Execution Time	Execution Time Normalized to C
v	50	10	100	5	500	1
w	200	3	400	1.5	600	1
x	250	2	500	1	500	1
y	400	2	800	1	800	1
z	5000	0.7	4100	0.8537	3500	1
Geometric Mean		2.4258		1.4497		1

10.3 Mathematical Preliminaries

- The real usefulness of the normalized geometric mean is that no matter which system us used as a reference, the ratio of the geometric means is consistent.
- This is to say that the ratio of the geometric means for System A to System B, System B to System C, and System A to System C is the same no matter which machine is the reference machine.

10.3 Mathematical Preliminaries

- The results that we got when using System B and System C as reference machines are given below.
- We find that 1.6733/1 = 2.4258/1.4497.

System A Execution Time	Execution Time Normalized to B	System B Execution Time	Execution Time Normalized to B	System C Execution Time	Execution Time Normalized to B
Geometric Mean	1.6733		1		0.6898

System A Execution Time	Execution Time Normalized to C	System B Execution Time	Execution Time Normalized to C	System C Execution Time	Execution Time Normalized to C
Geometric Mean	2.4258		1.4497		1

10.3 Mathematical Preliminaries

- The inherent problem with using the geometric mean to demonstrate machine performance is that all execution times contribute equally to the result.
- So shortening the execution time of a small program by 10% has the same effect as shortening the execution time of a large program by 10%.
  - Shorter programs are generally easier to optimize, but in the real world, we want to shorten the execution time of longer programs.
- Also, if the geometric mean is not proportionate. A system giving a geometric mean 50% smaller than another is not necessarily twice as fast!

10.3 Mathematical Preliminaries

- The *harmonic mean* provides us with the means to compare execution times that are expressed as a rate.
- The harmonic mean allows us to form a mathematical expectation of throughput, and to compare the relative throughput of systems and system components.
- To find the harmonic mean, we add the reciprocals of the rates and divide them into the number of rates:

$$H = n \div (1/x_1 + 1/x_2 + 1/x_3 + \ldots + 1/x_n)$$

10.3 Mathematical Preliminaries

- The harmonic mean holds two advantages over the geometric mean.
- First, it is a suitable predictor of machine behavior.
  - So it is useful for more than simply comparing performance.
- Second, the slowest rates have the greater influence on the result, so improving the slowest performance-- usually what we want to do-- results in better performance.
- The main disadvantage is that the geometric mean is sensitive to the choice of a reference machine.

10.3 Mathematical Preliminaries

- The chart below summarizes when the use of each of the performance means is appropriate.

Mean	Uniformly Distributed Data	Skewed Data	Data Expressed as a Ratio	Indicator of System Performance Under a Known Workload	Data Expressed as a Rate
Arithmetic	X			X	
Weighted Arithmetic				X	
Geometric		X	X		
Harmonic				X	X

10.3 Mathematical Preliminaries

- The objective assessment of computer performance is most critical when deciding which one to buy.
  - For enterprise-level systems, this process is complicated, and the consequences of a bad decision are grave.
- Unfortunately, computer sales are as much dependent on good marketing as on good performance.
- The wary buyer will understand how objective performance data can be slanted to the advantage of anyone giving a sales pitch.

10.3 Mathematical Preliminaries

- The most common deceptive practices include:
  - Selective statistics: Citing only favorable results while omitting others.
  - Citing only peak performance numbers while ignoring the average case.
  - Vagueness in the use of words like “almost,” “nearly,” “more,” and “less,” in comparing performance data.
  - The use of inappropriate statistics or “comparing apples to oranges.”
  - Implying that you should buy a particular system because “everyone” is buying similar systems.

Many examples can be found in business and trade journal ads.

10.4 Benchmarking

- Performance benchmarking is the science of making objective assessments concerning the performance of one system over another.
- *Price-performance ratios* can be derived from standard benchmarks.
- The troublesome issue is that there is no definitive benchmark that can tell you which system will run *your* applications the fastest for the least amount of money.

10.4 Benchmarking

- Many people erroneously equate CPU speed with performance.
- Measures of CPU speed include cycle time (MHz, and GHz) and millions of instructions per second (MIPS).
- Saying that System A is faster than System B because System A runs at 1.4GHz and System B runs at 900MHz is valid only when the ISAs of Systems A and B are identical.

10.4 Benchmarking

- In an effort to describe performance **independent of** clock speed and ISAs, a number of *synthetic benchmarks* have been attempted over the years.
- Synthetic benchmarks are programs that serve no purpose except to produce performance numbers.
- The earliest synthetic benchmarks, *Whetstone, Dhrystone,* and *Linpack* were relatively small programs that were easy to optimize.
- These programs are much too small to be useful in evaluating the performance of today's systems.

10.4 Benchmarking

- In 1988 the Standard Performance Evaluation Corporation (SPEC) was formed to address the need for objective benchmarks.
- SPEC produces benchmark suites for various classes of computers and computer applications.
- Their most widely known benchmark suite is the SPEC CPU benchmark.
- The SPEC2000 benchmark consists of two parts, CINT2000, which measures integer arithmetic operations, and CFP2000, which measures floating-point processing.

10.4 Benchmarking

- The SPEC benchmarks consist of a collection of kernel programs.
- These are programs that carry out the core processes involved in solving a particular problem.
  - Activities that do not contribute to solving the problem, such as I/O are removed.
- CINT2000 consists of 12 applications (11 written in C and one in C++); CFP2000 consists of 14 applications (6 FORTRAN 77, 4 FORTRAN 90, and 4 C).

A list of these programs can be found in Table 10.7 on Pages 467 - 468.

10.4 Benchmarking

- On most systems, more than two 24 hour days are required to run the SPEC CPU2000 benchmark suite.
- Upon completion, the execution time for each kernel (as reported by the benchmark suite) is divided by the run time for the same kernel on a Sun Ultra 10.
- The final result is the geometric mean of all of the run times.
- Manufacturers may report two sets of numbers: The *peak* and *base* numbers are the results with and without compiler optimization flags, respectively.

10.4 Benchmarking

- The SPEC CPU benchmark evaluates only CPU performance.
- When the performance of the entire system under high transaction loads is a greater concern, the *Transaction Performance Council* (TPC) benchmarks are more suitable.
- The current version of this suite is the TPC-C benchmark.
- TPC-C models the transactions typical of a warehousing business using terminal emulation software.

10.4 Benchmarking

- The TPC-C metric is the number of new warehouse order transactions per minute *(tpmC*), while a mix of other transactions is concurrently running on the system.
- The tpmC result is divided by the total cost of the configuration tested to give a price-performance ratio.
- The price of the system includes all hardware, software, and maintenance fees that the customer would expect to pay.

10.4 Benchmarking

- The *Transaction Performance Council* has also devised benchmarks for decision support systems (used for applications such as data mining) and for Web-based e-commerce systems.
- For all of its benchmarks, the systems tested must be available for general sale at the time of the test and at the prices cited in a full disclosure report.
- Results of the tests are audited by an independent auditing firm that has been certified by the TPC.

10.4 Benchmarking

- TPC benchmarks are a kind of simulation tool.
- They can be used to optimize system performance under varying conditions that occur rarely under normal conditions.
- Other kinds of simulation tools can be devised to assess performance of an existing system, or to model the performance of systems that do not yet exist.
- One of the greatest challenges in creation of a system simulation tool is in coming up with a realistic workload.

10.4 Benchmarking

- To determine the workload for a particular system component, system traces are sometimes used.
- Traces are gathered by using hardware or software probes that collect detailed information concerning the activity of a component of interest.
- Because of the enormous amount of detailed information collected by probes, they are usually engaged for a few seconds.
- Several trace runs may be required to obtain statistically useful system information.

10.4 Benchmarking

- Devising a good simulator requires that one keep a clear focus as to the purpose of the simulator
- A model that is too detailed is costly and time-consuming to write.
- Conversely, it is of little use to create a simulator that is so simplistic that it ignores important details of the system being modeled.
- A simulator should be validated to show that it is achieving the goal that it set out to do: A simple simulator is easier to validate than a complex one.

10.5 CPU Performance Optimization

- CPU optimization includes many of the topics that have been covered in preceding chapters.
  - CPU optimization includes topics such as pipelining, parallel execution units, and integrated floating-point units.
- We have not yet explored two important CPU optimization topics: Branch optimization and user code optimization.
- Both of these can affect performance in dramatic ways.

10.5 CPU Performance Optimization

- We know that pipelines offer significant execution speedup when the pipeline is kept full.
- Conditional branch instructions are a type of pipeline *hazard* that can result in flushing the pipeline.
  - Other hazards are include conflicts, data dependencies, and memory access delays.
- *Delayed branching* offers one way of dealing with branch hazards.
- With delayed branching, one or more instructions following a conditional branch are sent down the pipeline regardless of the outcome of the statement.

10.5 CPU Performance Optimization

```
        Add     R1, R2, R3
        Branch  Loop
        Div     R4, R5, RR6
Loop:   Mult    ...
```

This results in the following trace for the pipeline:

Time Slots:	1	2	3	4	5	6
Add	F	D	E			
Branch		F	D	E		
Div			F			
Mult				F	D	E

10.5 CPU Performance Optimization

Time Slots:	1	2	3	4	5	6
Branch	F	D	E			
Add		F	D	E		
Mult			F	D	E	

10.5 CPU Performance Optimization

- The responsibility for setting up delayed branching most often rests with the compiler.
- It can choose the instruction to place in the delay slot in a number of ways.
- The first choice is a useful instruction that executes regardless of whether the branch occurs.
- Other possibilities include instructions that execute if the branch occurs, but do no harm if the branch does not occur.
- Delayed branching has the advantage of low hardware cost.

10.5 CPU Performance Optimization

- Branch prediction is another approach to minimizing branch penalties.
- *Branch prediction* tries to avoid pipeline stalls by guessing the next instruction in the instruction stream.
  - This is called *speculative execution.*
- Branch prediction techniques vary according to the type of branching. If/then/else, loop control, and subroutine branching all have different execution profiles.

10.5 CPU Performance Optimization

There are various ways in which a prediction can be made:

- *Fixed predictions* do not change over time.
- *True predictions* result in the branch being always taken or never taken.
- *Dynamic prediction* uses historical information about the branch and its outcomes.
- *Static prediction* does not use any history.

10.5 CPU Performance Optimization

- When fixed prediction assumes that a branch is not taken, the normal sequential path of the program is taken.
- However, processing is **done in parallel** in case the branch occurs.
- If the prediction is correct, the preprocessing information is deleted.
- If the prediction is incorrect, the speculative processing is deleted and the preprocessing information is used to continue on the correct path.

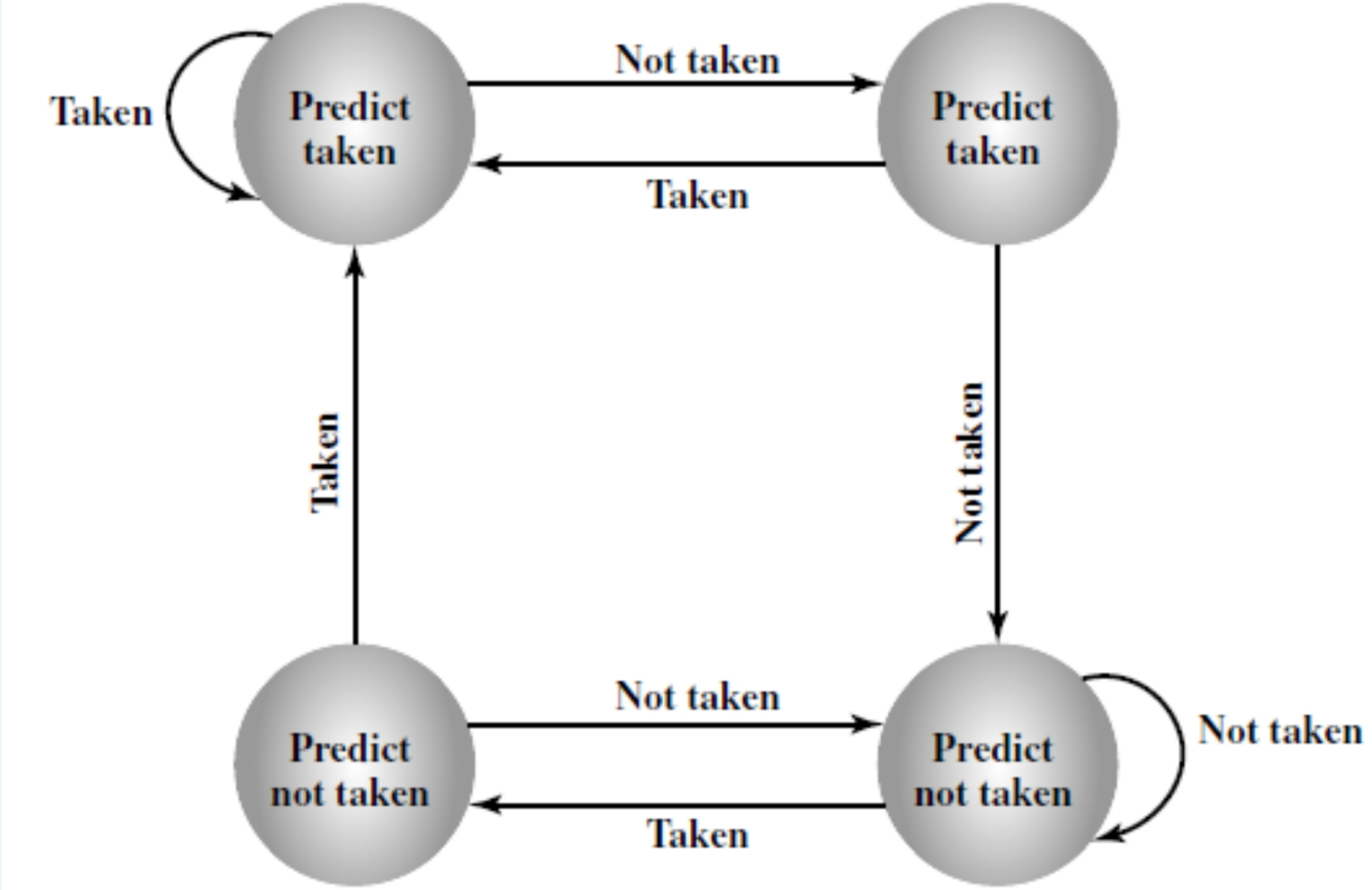
10.5 CPU Performance Optimization

- When fixed prediction assumes that a branch is always taken, state information is saved before the speculative processing begins.
- If the prediction is correct, the saved information is deleted.
- If the prediction is incorrect, the speculative processing is deleted and the saved information is restored allowing execution to continue to continue on the correct path.

10.5 CPU Performance Optimization

- Dynamic prediction employs a high-speed *branch prediction buffer* to combine an instruction with its history.
- The buffer is indexed by the lower portion of the address of the branch instruction that also contains extra bits indicating whether the branch was recently taken.
  - *One-bit dynamic prediction* uses a single bit to indicate whether the last occurrence of the branch was taken.
  - *Two-bit branch prediction* retains the history of the previous to occurrences of the branch along with a probability of the branch being taken.

10.5 CPU Performance Optimization



10.5 CPU Performance Optimization

- The earliest branch *prediction implementations used static branch prediction*.
- Most newer processors (including the Pentium, PowerPC, UltraSparc, and Motorola 68060) use two-bit dynamic branch prediction.
- Some superscalar architectures include branch prediction as a user option.
- Many systems implement branch prediction in specialized circuits for maximum throughput.

10.5 CPU Performance Optimization

- The best hardware and compilers will never equal the abilities of a human being who has mastered the science of effective algorithm and coding design.
- People can see an algorithm in the context of the machine it will run on.
  - For example a good programmer will **access a stored column-major array in column-major order**.
- We end this section by offering some tips to help you achieve optimal program performance.

10.5 CPU Performance Optimization

- *Operation counting* can enhance program performance.
- With this method, you count the number of instruction types executed in a loop then determine the number of machine cycles for each instruction.
- The idea is to provide the best mix of instruction types for a particular architecture.
- Nested loops provide a number of interesting optimization opportunities.

10.5 CPU Performance Optimization

- *Loop unrolling* is the process of expanding a loop so that each new iteration contains several of the original operations, thus performing more computations per loop iteration. For example:

```
for (i = 1; i <= 30; i++)
   a[i] = a[i] + b[i] * c;
```

becomes

```
for (i = 1; i <= 30; i+=3)
  { a[i] = a[i] + b[i] * c;
    a[i+1] = a[i+1] + b[i+1] * c;
    a[i+2] = a[i+2] + b[i+2] * c; }
```

10.5 CPU Performance Optimization

- *It reduces loop overhead*
- *Helps with control hazards in pipelines*
- *It typically enables operations to execute in parallel*

10.5 CPU Performance Optimization

- *Loop fusion* combines loops that use the same data elements, possibly improving cache performance. For example:

```
for (i = 0; i < N; i++)
     C[i] = A[i] + B[i];
for (i = 0; i < N; i++)
     D[i] = E[i] + C[i];
```

becomes

```
for (i = 0; i < N; i++)
  { C[i] = A[i] + B[i];
    D[i] = E[i] + C[i]; }
```

10.5 CPU Performance Optimization

```
for (i=1; i<100; i++)
    A[i] = B[i] + 8;
for (i=1; i<99; i++)
    C[i] = A[i+1] * 4;
```

becomes

```
A[1] = B[1] + 8;
for (i=2; i<100; i++)  {
    A[i] = B[i] + 8;}
for (i=1; i<99; i++)  {
    C[i] = A[i+1] * 4; }
```

```
i = 1;
A[i] = B[i] + 8;
for (j=0; j<98; j++)  {
    i = j + 2;
    A[i] = B[i] + 8;
    i = j + 1;
    C[i] = A[i+1] * 4; }
```

10.5 CPU Performance Optimization

- *Loop fission* splits large loops into smaller ones to **reduce data dependencies and resource conflicts**.
- A loop fission technique known as *loop peeling* removes the beginning and ending loop statements. For example:

```
for (i = 1; i < N+1; i++)
{ if (i==1)
      A[i] = 0;
  else if (i == N)
      A[i] = N;
  else A[i] = A[i] + 8; }
```

becomes

```
A[1] = 0;
for (i = 2; i < N; i++)
  A[i] = A[i] + 8;
A[N] = N;
```

10.6 Disk Performance

- Optimal disk performance is critical to system throughput.
- Disk drives are the slowest memory component, with the fastest access times one million times longer than main memory access times.
- A slow disk system can choke transaction processing and drag down the performance of all programs when virtual memory paging is involved.
- Low CPU utilization can actually indicate a problem in the I/O subsystem, because the CPU spends more time waiting than running.

10.6 Disk Performance

- Disk utilization is the measure of the percentage of the time that the disk is busy servicing I/O requests.
- It gives the probability that the disk will be busy when another I/O request arrives in the disk service queue.
- Disk utilization is determined by the speed of the disk and the rate at which requests arrive in the service queue. Stated mathematically:

$$\text{Utilization} = \text{Request Arrival Rate} \div \text{Disk Service Rate}.$$

where the arrival rate is given in requests per second, and the disk service rate is given in I/O operations per second (IOPS)

10.6 Disk Performance

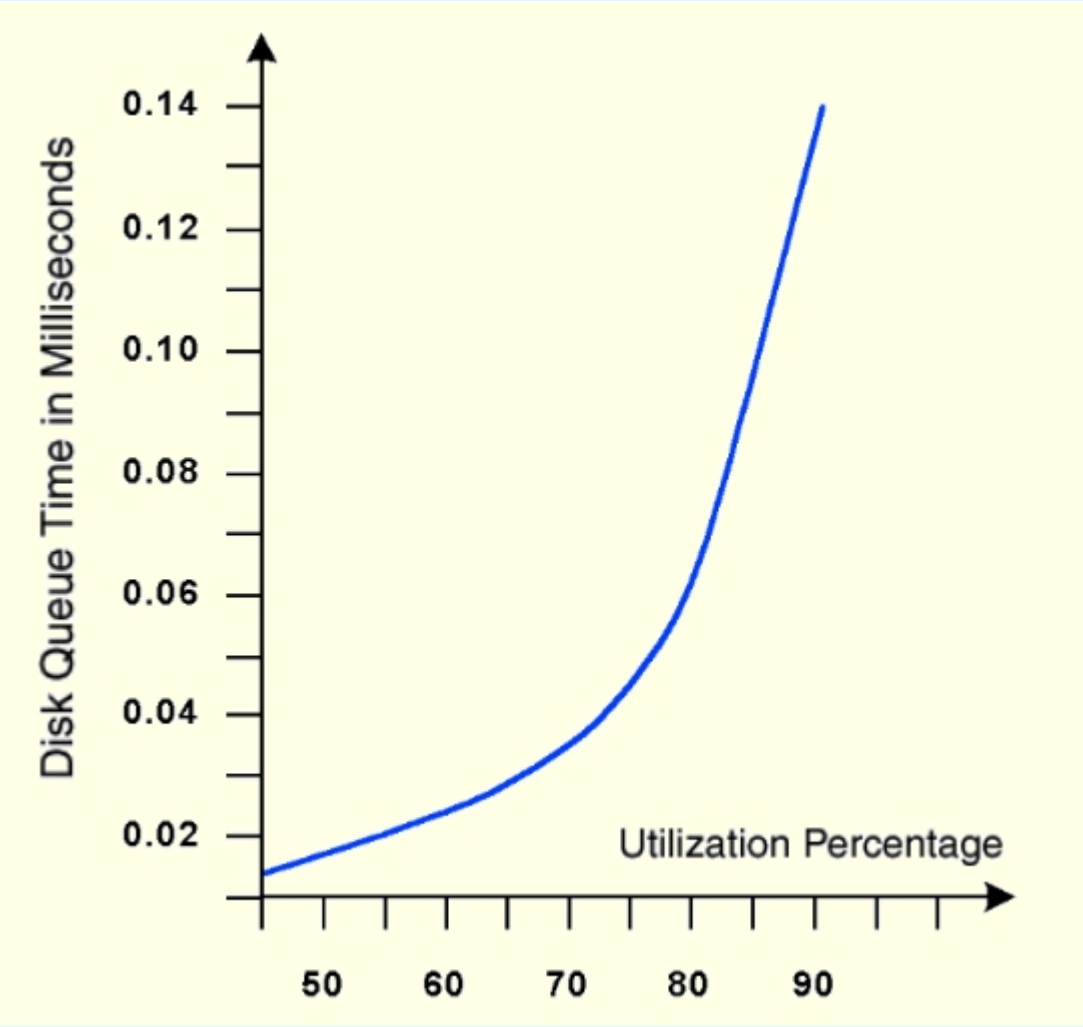
- The amount of time that a request spends in the queue is directly related to the service time and the probability that the disk is busy, and it is indirectly related to the probability that the disk is idle.
- In formula form, we have:

$$\text{Time in Queue} = (\text{Service time} \times \text{Utilization}) \div (1 - \text{Utilization})$$

- The important relationship between queue time and utilization (from the formula above) is shown graphically on the next slide.

10.6 Disk Performance

The “knee” of the curve is around 78%. This is why 80% is the rule-of-thumb upper limit for utilization for most disk drives.
Beyond that, queue time quickly becomes excessive.



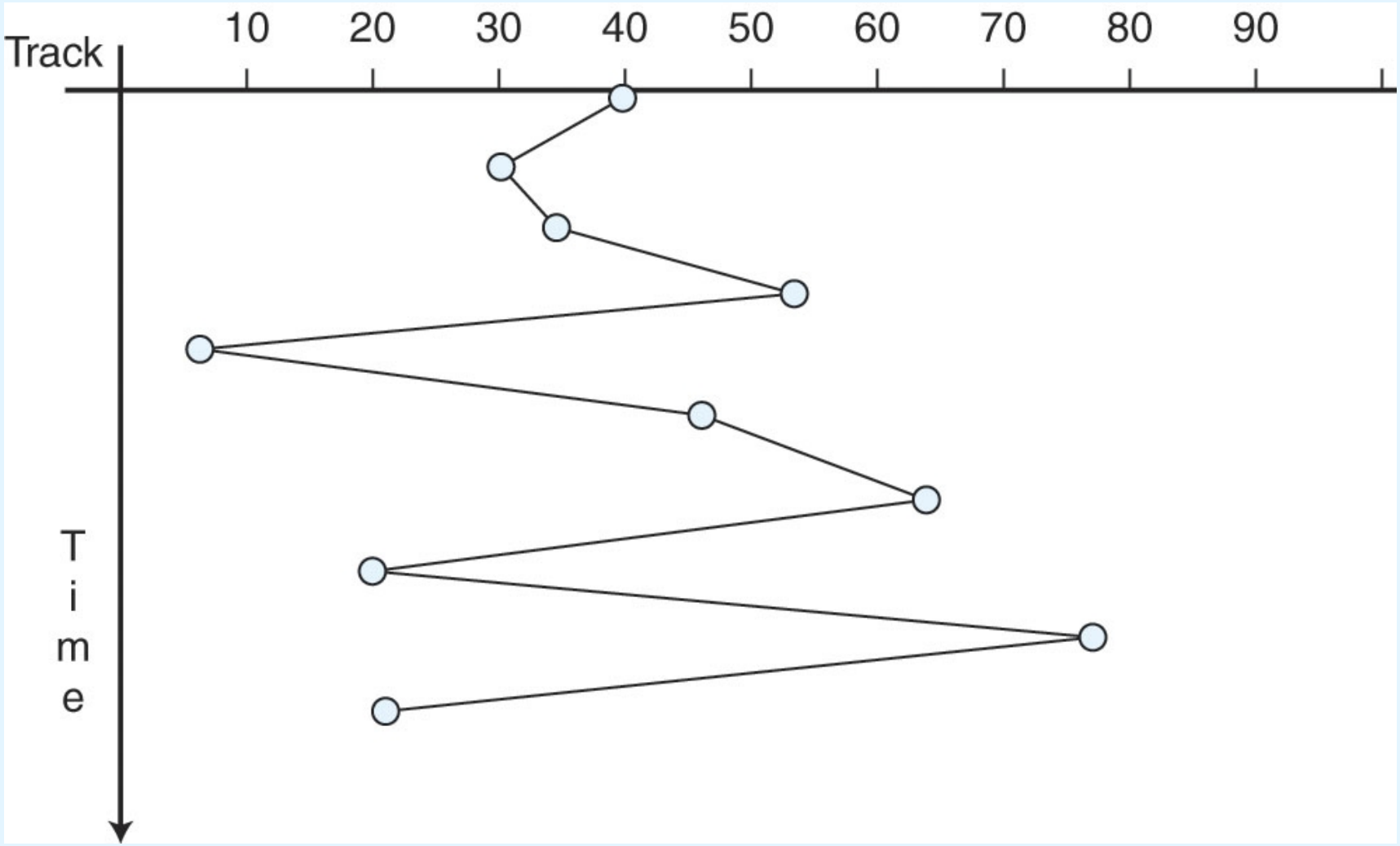
10.6 Disk Performance

- The manner in which files are organized on a disk greatly affects throughput.
- Disk arm motion is the greatest consumer of service time.
- Disk specifications cite average seek time, which is usually in the range of 5 to 10ms.
- However, a full-stroke seek can take as long as 15 to 20ms.
- Clever disk scheduling algorithms endeavor to minimize seek time.

10.6 Disk Performance

- The most naïve disk scheduling policy is *first-come, first-served* (*FCFS*).
- As its name implies, FCFS services all I/O requests in the order in which they arrive in the queue.
- With this approach, there is no real control over arm motion, so random, wide sweeps across the disk are possible.
- Performance is unpredictable and variable according to shifts in workload.

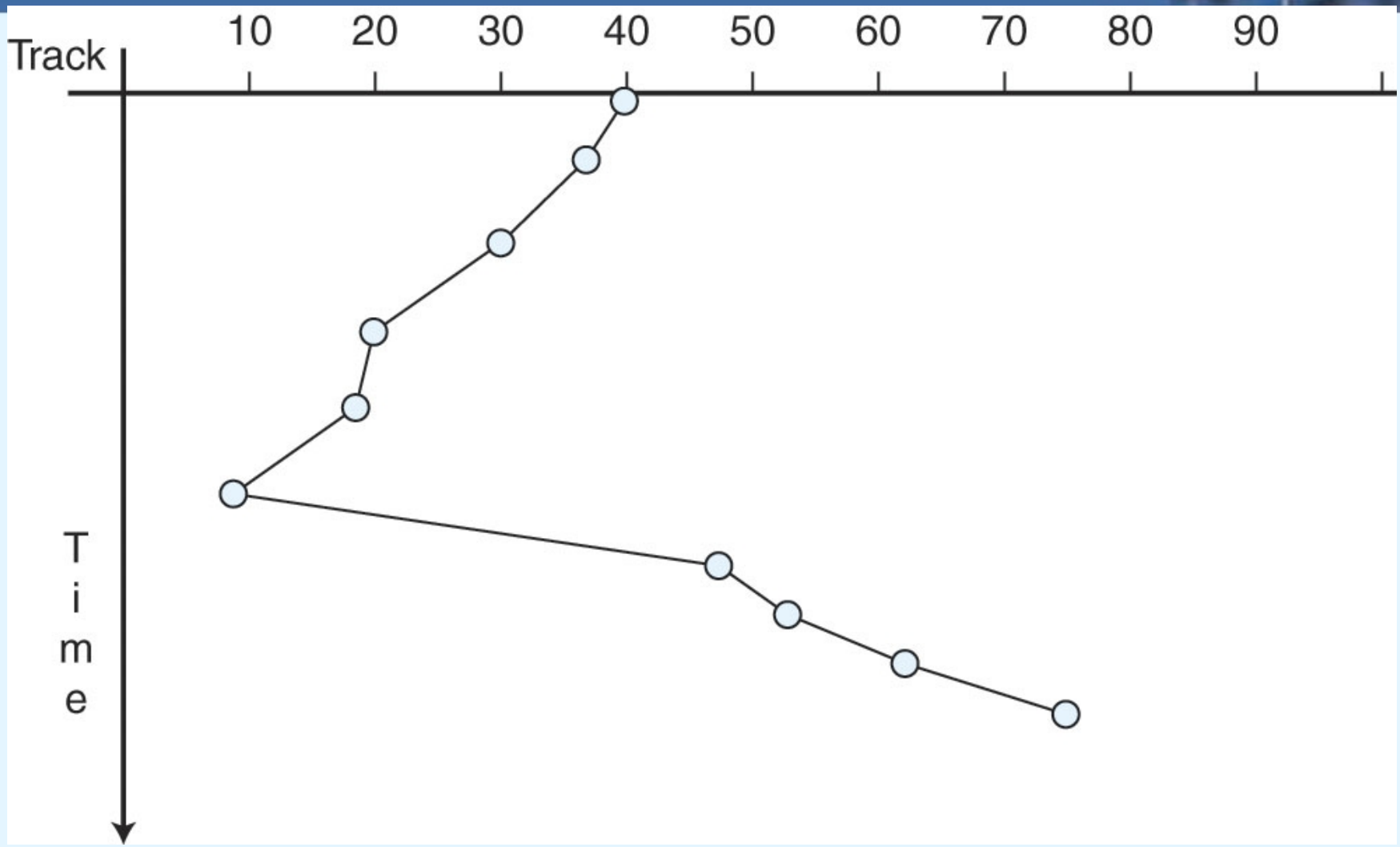
10.6 Disk Performance



10.6 Disk Performance

- Arm motion is reduced when requests are ordered so that the disk arm moves only to the track nearest its current location.
- This is the idea employed by the *shortest seek time first* (*SSTF*) scheduling algorithm.
- With SSTF, starvation is possible: A track request for a "remote" track could keep getting shoved to the back of the queue nearer requests are serviced.
  - Interestingly, this problem is at its worst with low disk utilization rates.

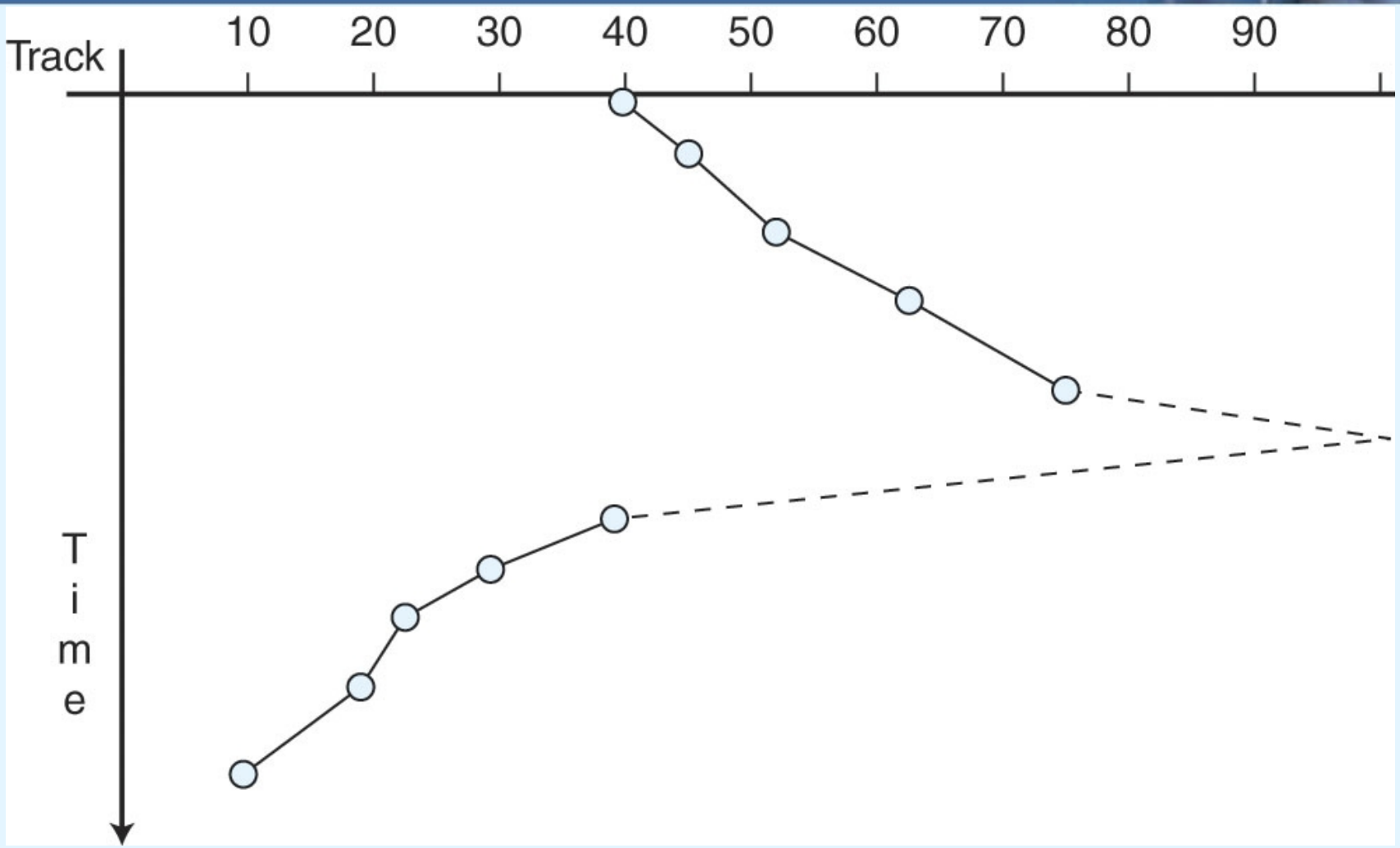
10.6 Disk Performance



10.6 Disk Performance

- To avoid the starvation in SSTF, fairness can be enforced by having the disk arm continually sweep over the surface of the disk, stopping when it reaches a track for which it has a request.
  - This approach is called an *elevator algorithm*.
- In the context of disk scheduling, the elevator algorithm is known as the SCAN (which is *not* an acronym).
- While SCAN entails a lot of arm motion, the motion is constant and predictable.

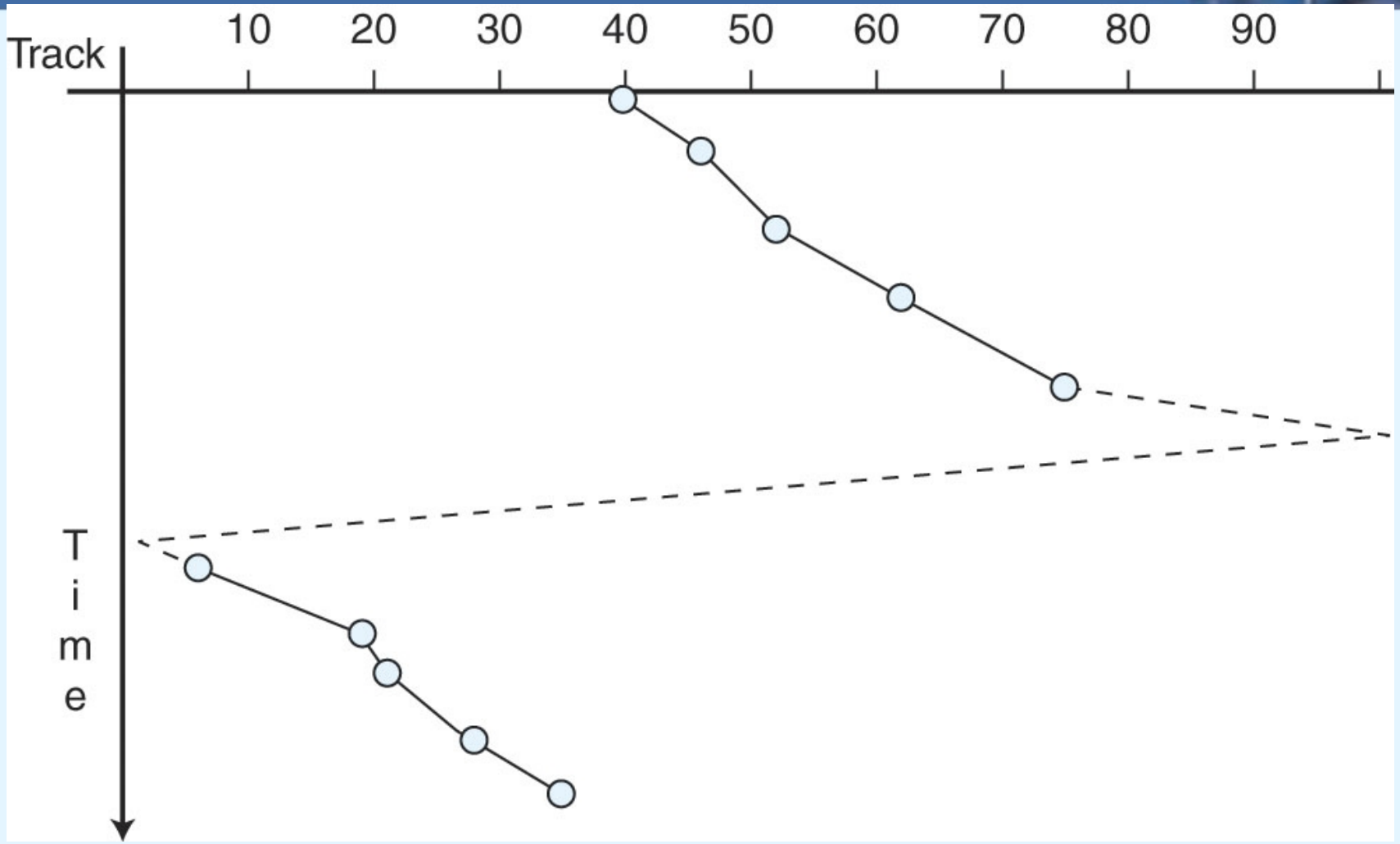
10.6 Disk Performance



10.6 Disk Performance

- A SCAN variant, called *C-SCAN* for circular SCAN, treats track zero as if it is adjacent to the highest-numbered track on the disk.
- The arm moves in one direction only, providing a simpler SCAN implementation.
- The disk arm motion of SCAN and C-SCAN is can be reduced through the use of the *LOOK* and *C-LOOK* algorithms.
- Instead of sweeping the entire disk, they travel only to the highest- and lowest-numbered tracks for which they have requests.

10.6 Disk Performance



10.6 Disk Performance

- At high utilization rates, SSTF performs slightly better than SCAN or LOOK. But the risk of starvation persists.
- Under very low utilization (under 20%), the performance of any of these algorithms will be acceptable.
- No matter which scheduling algorithm is used, **file placement** greatly influences performance.
- When possible, the most frequently-used files should reside in the center tracks of the disk, and the disk should be periodically defragmented.

10.6 Disk Performance

- The best way to reduce disk arm motion is **to avoid using the disk as much as possible**.
- To this end, many disk drives, or disk drive controllers, are provided with cache memory or a number of main memory pages set aside for the exclusive use of the I/O subsystem.
- Disk cache memory is usually associative.
  - Because associative cache searches are time-consuming, performance can actually be better with smaller disk caches because hit rates are usually low.

10.6 Disk Performance

- The best way to reduce disk arm motion is to avoid using the disk as much as possible.
- To this end, many disk drives, or disk drive controllers, are provided with cache memory or a number of main memory pages set aside for the exclusive use of the I/O subsystem.
- Disk cache memory is usually associative.
  - Because associative cache searches are time-consuming, performance can actually be better with smaller disk caches because hit rates are usually low.

10.6 Disk Performance

- Many disk drive-based caches use *prefetching* techniques to reduce disk accesses.
- When using prefetching, a disk will **read a number of sectors** subsequent to the one requested with the expectation that one or more of the subsequent sectors will be needed “soon.”
- Empirical studies have shown that over 50% of disk accesses are sequential in nature, and that prefetching increases performance by 40%, on average.

10.6 Disk Performance

- Prefetching is subject to *cache pollution*, which occurs when the cache is filled with data that no process needs, leaving less room for useful data.
- Various replacement algorithms, LRU, LFU and random, are employed to help keep the cache clean.
- Additionally, because disk caches serve as a staging area for data to be written to the disk, some disk cache management schemes evict all bytes after they have been written to the disk.

10.6 Disk Performance

- With cached disk writes, we are faced with the problem that cache is volatile memory.
- In the event of a massive system failure, data in the cache will be lost.
- An application believes that the data has been committed to the disk, when it really is in the cache. If the cache fails, the data just disappears.
- To defend against power loss to the cache, some disk controller-based caches are mirrored and supplied with a battery backup.

10.6 Disk Performance

- Another approach to combating cache failure is to employ a write-through cache where a copy of the data is retained in the cache in case it is needed again “soon,” but it is simultaneously written to the disk.
- The operating system is signaled that the I/O is complete only after the data has actually been placed on the disk.
- With a write-through cache, performance is somewhat compromised to provide reliability.

Chapter 10 Conclusion

- Computer performance assessment relies upon measures of central tendency that include the arithmetic mean, weighted arithmetic mean, the geometric mean, and the harmonic mean.
- Each of these is applicable under different circumstances.
- Benchmark suites have been designed to provide objective performance assessment. The most well respected of these are the SPEC and TPC benchmarks.

Chapter 10 Conclusion

- CPU performance depends upon many factors.
- These include pipelining, parallel execution units, integrated floating-point units, and effective branch prediction.
- User code optimization affords the greatest opportunity for performance improvement.
- Code optimization methods include loop manipulation and good algorithm design.

Chapter 10 Conclusion

- Most systems are heavily dependent upon I/O subsystems.
- Disk performance can be improved through good scheduling algorithms, appropriate file placement, and caching.
- Caching provides speed, but involves some risk.
- Keeping disks defragmented reduces arm motion and results in faster service time.