Performance

Objectives

Producing a correct result is not the only goal of software: it must be produced within a reasonable timeframe, otherwise users may not wait for a response and just give up. Providing a timely response is made more difficult by the interactions between software components and between software and hardware, as well as the inherent conflicts between different definitions of performance. As a software engineer, you will need to find out what kind of performance matters for your code, measure the performance of your code, define clear and useful performance goals, and ensure the code meets these goals. You will need to keep performance in mind when designing and writing code, and to "debug" the performance issues that cause your software to not meet its performance goals.

In this module, you will learn:

- How to define and quantify performance

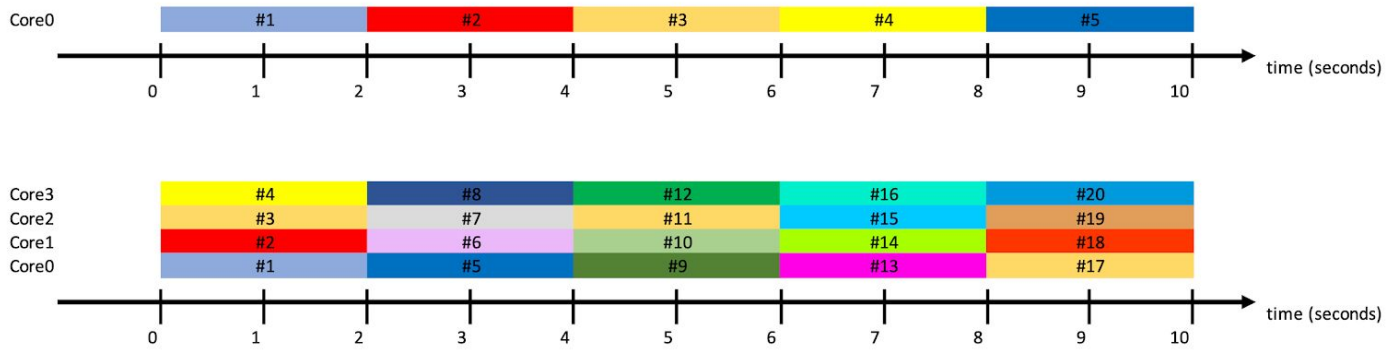
- What to measure and how to measure performance
- How to set useful performance goals
- How to improve software performance

What is performance?

Performance is the measure of efficiency. Intuitively, a "high performance" system can perform tasks faster than a "low performance" one. This lecture is all about refining that intuition into a clearer definition and helping you measure and improve the performance of your software.

There are two facets to the performance of a software system: *throughput* and *latency*. Throughput is the number of work items passing through the system per unit of time, such as the number of requests a Web server can process per second. Latency is the time it takes for a single work item to be processed, such as the time between a Web server receiving an HTTP request and that same machine receiving the response from the server.

The relationship between these two types depends on the system. If the system is sequential, then throughput is the inverse of latency. That is, if each item takes N seconds to be processed, then the system can process 1/N items per second. But if the system contains parallelism, then the throughput may be higher than that limit. For instance, a Web server running on 4 cores can serve 4/N requests per second if the latency on each core is N and there is no contention in the system.



Performance of software systems is typically not constant; instead, it is a *distribution*. For instance, some requests may be simpler than others, and will thus be processed faster. The various levels of caching in a modern system can also introduce differences in the processing time of identical requests. For instance, if all the data needed to process a request is already in the CPU cache, it will be faster to process it than the previous request that required the CPU to bring in the data from main memory.

To reason about distributions, humans like to reduce them to a few key metrics, such as the *tail* of that distribution. Measures such as the mean or median are often not useful because humans want guarantees from systems. For instance, "letters should be delivered by the post in 3 days or less at least 97% of the time" is a guarantee people care about more than what the average delivery time is, which is likely why the Swiss Federal Council chose it. Thus, it is often more important to improve a system's performance tail, such as its 99th percentile, than its average performance. This is particularly important in systems that depend on many components; if processing one item requires calling 100 components, then there is a >63% chance that at least one component will have a performance worse than its 99th percentile for that item. See The Tail at Scale on a discussion of responsiveness in Web services.

One key aspect of software design is to define its performance targets, not only in terms of distribution, but in terms of *metrics* and *workloads*. For instance, should a Web server be measured in terms of requests per second or bytes per second? If measuring requests, what is a typical workload to use for measurements?

We have all seen semantic interfaces, i.e., descriptions of what a particular does (such as the documentation for `java.util.Map`). Deploying, and using a software system, however, also requires knowledge of its performance properties in addition to its semantic behavior. In fact, today's systems are often considered to be working "correctly" only if they meet certain performance targets, as will be discussed below, since not meeting these targets can have serious consequences. Unfortunately, there is no widely used equivalent performance description that engineers can use to decide if a system meets their performance requirements. Consequently, engineers either rely on certain incomplete outlines of performance (e.g., "this code runs in O(1) time") or are forced to come up with their own understanding of the system's performance by benchmarking the system. Recent research on the topic has proposed the notion of performance contracts, but a lot remains to be done.

How is software performance measured?

> "When you can measure what you are speaking about, and express it in numbers, you know something about it. When you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."
>
> William Thomson, 1st Baron Kelvin

There are multiple ways to measure performance, ranging from the most empirical (via benchmarking) to the most theoretical (via modeling), with a good intermediate point (via profiling).

Benchmarking

Estimating software performance via direct measurement is called *benchmarking*. This is conceptually simple: write down the current time, execute an operation, write down the current time, then subtract the two times to know how long the operation took.

However, there are many caveats in practice, and one has to be careful to not fool themselves through direct measurements. Are you measuring the right thing? Are you doing it in the right circumstances? Are you doing it in the right way?

For instance, measuring time itself is not trivial. Some methods are faster than others, such as obtaining the time from the CPU directly rather than going through the operating system's timezone-handling logic. One also cannot rely on measuring a single operation, due to likely variance in the distribution of operation times, thus a benchmark needs to run the operation enough times to get statistically meaningful results. Some run times have warmup effects, such as tiered just-in-time compilation in the Java runtime, meaning that the first few times a piece of code is run will be slower than the next ones.

There are two general categories of benchmarks: *end-to-end benchmarks*, which measure the performance of an entire system, and *microbenchmarks*, which measure the performance of a small part of a system, such as a single function or class. The type of benchmark is often determined by the kind of "workload" you choose, i.e., the mix of requests. For instance, when measuring the performance of a filesystem, are you only doing reads, or are you also doing writes? This can make a big difference, as it hits different parts of the system.

End-to-end benchmarks are more representative of what you'd expect from the system as a whole, especially if you know what kind of inputs are typical, but they are more complex to set up and run, and their results are harder to interpret. Microbenchmarks allow you to precisely know how fast a specific part of your code is on some inputs, at the risk of over-interpreting these results and optimizing the wrong thing. Micro-benchmarks also have a higher risk of not measuring what you think they measure due to compiler optimizations; for instance, a common microbenchmarking mistake is to execute a side-effect-free operation in a loop without doing anything with the

operation's result, thereby allowing the compiler to surreptitiously optimize your code by removing the loop entirely – you end up benchmarking a very fast and empty piece of code.

There are a variety of benchmarking frameworks you can use for your software, such as JMH for Java or BenchmarkDotNet for C#. Note how both of these examples are developed by the same organizations that develop the languages they target; this is not a coincidence, writing a good benchmark framework requires deep knowledge of the platform. There are simply too many factors to take into account; as a software engineer, you should avoid rolling your own benchmark framework unless you are an expert. Even for one-off benchmarks, rolling your own "simple" code can and will mislead you because the results are unlikely to be what you had in mind.

Benchmark results are only meaningful if you compare them to *baselines*, to know how much you improved and how much further you could go. If you are improving on existing code, which you should have before spending time on optimization, you can use the existing code as a baseline to know how much faster your new code is. Ideally you'd like your software to run in zero time, but this is infeasible, so you can write a baseline that does less work than you need to get a lower bound. For instance, a lower bound baseline for a sorting algorithm could be a loop that reads every item in the sequence to be sorted but does nothing else.

> *Doing everything right and getting wrong results*
>
> It is unfortunately entirely possible to write a perfectly reasonable-looking benchmark and still obtain incorrect results, because of unexpected implementation details. For instance, look at this issue filed against the .NET framework, in which a user points out that .NET's implementation of a hash map is 5-10x slower than Java's. It turns out, after more investigation, that this effect is entirely due to the way the user was generating inputs for the benchmark. The inputs happen to produce sequential hash codes in Java, which allows Java's hash map to efficiently use the CPU cache, whereas .NET's hash code algorithm works differently and does not produce sequential hash code for these inputs.
>
> For a more systematic look at this, see Producing Wrong Data Without Doing Anything Obviously Wrong!, Mytkowicz et al., ASPLOS 2009.

Profiling

If your system is not as fast as you need it to be, the next question is *why*. A widely held belief about programs is known as the 80/20 rule (or Pareto principle) and claims that 80% of the resource consumption (CPU time, memory, I/O bandwidth, etc.) comes from 20% of the program, so it make sense to focus optimization efforts on that 20%. How do you know which the 20% is?

The answer is obtained by *profiling*. By building a performance profile of your software, you will know which part is the *bottleneck* that slows the system down. It is crucial to obtain a performance profile before benchmarking specific operations or attempting to improve your code's performance; if you do not know why the software is slow, you will likely waste time optimizing operations that are already fast enough. For instance, reducing the cost of serializing a request into XML from 50ms down to 10ms looks impressive, but it is pointless if the software then spends 10s waiting for the response.

There are two main ways to profile software: *instrumentation* and *sampling*. Instrumenting a piece of code means modifying it to add code that keeps track of what is happening, which provides detailed and accurate information at the cost of slowing down the code. Sampling a piece of code means periodically pausing it while it runs and checking which part of the code is executing, which has less overhead but is less accurate. In general, sampling is a better idea if the code is likely to have a few big bottlenecks, since it will not hide them with profiling overhead, while instrumenting works better on code that has a lot of small tasks that add up. Note that profiling is not limited to measuring time; it can also measure memory use, for instance, and even keep track of how many of each kind of objects are in memory. The VisualVM tool for Java can do both.

While working on a project, it is easy to forget the performance and concentrate on the functionality. Doing so can lead to bad effects on your product popularity and rentability. Performance can be crucial to your product's success and it must not drop when a new version comes out. A solution to this problem is to have a test suite to run regularly (the regularity depends on the time required to run it, it can be on every commit, every night or once in a week for example). This test suite must measure some performance metrics and produce results you can analyse. In this way, you can see how the performance evolves during the development and detect any drop. If the performance drops at some point, the results of the test suite run will show it and it will be easy to spot what changes in the code induced it. Therefore, it is easier to correct the performance problem. The metrics used must be user oriented, that means they must reflect the user experience. For a web page, the metrics could be the latency of the request, the time needed to load the content of the page, the time to wait to checkout an order.

Modeling

Another way to analyze a system's performance is *modeling*. Instead of measuring the real system's performance, one can build a high-level model of it instead and compute theoretical limits on the system's performance. This can be done before any code is written, which avoids wasting time building a system that is slow by design. For instance, if the system interacts with an external component that is known to be slow, modeling different options for this interaction can help engineers decide which one they should use.

Using such models helps to predict the speedup achievable by a given change in the program or the theoretical maximum speedup you can achieve. This is useful to know where you need to put your effort. One such model is *Amdahl's law*. One of its intuitive interpretations is the following: "A program cannot be faster than the slowest of its components". It is often used in parallel computing and, in this context, it means that if a program needs 10 hours to complete in a single thread, and that 1 hour of this work cannot be parallelized, then no matter how much parallelization is added, it would never take less than 1 hour to complete. Here is the formal definition of the law for the parallel computing:

$$S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

Where

- $S_{latency}$ is the speedup of the whole task.
- s is the number of threads in the parallel version.
- p is the portion of the original execution time that will benefit from the parallelization.

By taking the limit of the speedup with s goes to infinity, we obtain the maximum theoretical speedup achievable. For our previous example, we obtain a maximum speedup of $\frac{1}{1-0.9} = 10$.

Another interpretation of this law gives a good point to keep in mind while working on performance: "improving the performance of a part of the program that takes x% of the total execution time, can only improve the total execution time by x%". This is quite intuitive when read but is easy to forget while working on a real project. Let's say you want to improve the performance of a SQL request that takes quite long but represents only 5% of the total execution time of one user request for your web server. Even if you work with a team of 10 people during 2 months to optimize it and, let's say, you manage to reduce to an infinitesimal amount the execution time of this request (which is of course impossible), you would improve to the total latency by only 5%. Therefore, you should always analyse what proportion of the total execution time a task represents before optimizing it: if it is too small, your hard work will have negligible benefits on the final performance of the product and you would have wasted your time and effort.

What is good performance?

It is tempting to think that all systems should be as fast as possible, and that any software that is slower than it could be is a bug. However, performance is only one aspect of a system, and software engineers have deadlines to meet. Thus, systems should be *fast enough*, not perfect. What "enough" is depends on the context.

Some software systems have specific performance requirements. These are often called *Service Level Objectives*, or SLOs for short, which are part of a *Service Level Agreement*, or SLA for short. For instance, an SLO could be "the system must handle at least 1000 requests per second", or "99% of requests must be served in less than 2 seconds". This can be measured while the system is running, and violations are breaches of contract that may cost money. See Nines are Not Enough: Meaningful Metrics for Clouds for more thoughts on this subject.

Software without explicit performance requirements still has some, but they are implicit, making them harder to define. For instance, in 2006 Google found that traffic dropped 20% if their search results took 0.9s to show instead of 0.4s. In 2017, Akamai found that even a 100ms increase in page load time could lead to a 7% drop in sales. In general, if users feel that an interface is "too slow", they won't like it, and if they have another choice they may use it instead.

How can you improve performance?

If your software is not fast enough, there are many ways to improve its performance, from high-level design choices to low-level machine-specific tweaks. High-level choices are good to keep in mind at all times as they also generally result in more maintainable code. Low-level optimizations should only be performed if you have a reason to believe it is worth it, because such optimizations make code less maintainable and, as a result, any performance gain now might be anyway erased in the future due to the difficulty of refactoring your code.

> "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."
>
> Donald Knuth (source)

High-level choices

We will first see some high-level design choices. These can be applied at any level of a software system, including in individual functions. They should be your main tools to improve performance, as the gains from these choices are likely to dwarf low-level optimizations in most codebases. Most of them revolve around how data flows in a system.

In general, you should keep these choices in mind when designing software, so that it can be optimized in the future. Don't make your system more complex than it has to be to meet its present requirements, but don't accidentally make design decisions that prevent further optimizations when the requirements change, either.

A common property of all these is that neither of them are necessary for correctness: whether you cache, batch, parallelize, speculate, etc. should make no difference to whether the tasks are done correctly. It will only make a difference in the latency or throughput of performing the tasks.

Caching

The fastest way to perform an operation is to not do it at all. You can achieve this whenever you can reuse a result you computed previously – this is called *caching*. Caching applies to any kind of computation, ranging from compute-heavy to I/O-heavy. Sometimes only part of the operation can be reused, but this is typically better than redoing everything.

Your modules that perform expensive operations, such as making requests to external services or computing expensive analytics, should be designed with caching and incrementality in mind. For instance, instead of one Web request to obtain pieces of data with very different expiration dates, such as the latest chat messages and the user's profile information, split it in multiple requests that can be cached independently.

Caching is a good use of the Decorator design pattern: instead of mixing the code that deals with requests and responses and the caching logic, write a decorator that caches responses, which can then be used regardless of the actual request-response logic.

Here is an example of performance design patterns that Amazon recommends for their own Web services.

> Caching is everywhere in software apps. Let us do a simple manipulation. Take your phone, turn off the Wi-Fi and mobile data and try to launch your favorite social media app, preferably one that has a feed and images. You'll notice that even though there's no connection to the internet, there is still some content there, you can even scroll through it.
>
> The following steps can be done for Android users:
>
> While still having the Wi-Fi and mobile data off, go to settings -> apps -> your social media app -> storage and clear its cache. Now try to open the app again. You'll notice that the feed is wiped (or at least there are no images anymore).
>
> In the exercise set, you'll have the chance to write some code that performs queries and uses caching to improve performance.

Batching

Whenever software performs an operation, there is a certain amount of overhead associated with processing the request for that operation, just like when you call a plumber to fix your sink, there is the overhead of loading his tools in the van, driving the van to your apartment, then cleaning up, driving back, etc. As a computing-oriented example, a system might need to reset resources before every operation, for isolation purposes, or a remote call might be processed quickly on the remote server but take time to get there and back. If the cost of a request is large enough compared to executing an operation, it makes sense to *batch* multiple operations together in a single request, i.e., pay the overhead once and process multiple operations in one shot. This is akin to the plumber coming to your building and fixing the sinks in multiple apartments with one trip.

Batching is feasible as long as the system's abstraction does not require all operations to be executed immediately. For instance, the system can hold a buffer of requested operations and send them all in one request if the user has not requested any additional operations in a few seconds. If your sink can wait until after your neighbor's sink is fixed, then the plumber can batch the fixing of sinks.

Batching can even be done with operations that depend on each other, if the dependencies are in well-known patterns. The Cap'n Proto framework for remote procedure calls implements this, jokingly calling it "time travel".

> Here is an example in SQL with Python-like pseudocode..
> Consider the following list of transactions:
>
> ```
> list_of_operations = ["insert MyTable values ('new value', 1)",
> ```

```
                        "update MyTable set mytext = 'updated text' where id = 2",
                        "update MyTable set mytext = 'updated text' where id = 3"]
```

One can iterate over the elements of the list committing each element at a time, as follows:

```
open connection conn
foreach operation in list_of_operations:
    cmd = new SqlCommand(operation, conn)
    cmd.Execute()
```

A more performant solution is to use batching. There are multiple batching strategies. One of them is wrapping the whole thing in one single transaction, as follows:

```
open connection conn
// wraps the whole thing into a transaction

SqlTransaction transaction = conn.BeginTransaction();
foreach operation in list_of_operations:
    cmd = new SqlCommand(operation, conn)
    cmd.Execute()
transaction.Commit() // commits the transaction
```

Parallelism

Tasks that are independent of each other can be executed in parallel if there are enough resources available. This is not only the case for compute-heavy tasks, but also for I/O-bound tasks: while waiting for a response, a single CPU core can execute other operations. Some standard libraries make this particularly easy, such as Java's parallel streams, C#'s PLINQ, and Spark for Scala, Java and Python.

Parallelism is made easy if operations do not depend on each other. While it is possible to execute operations in parallel even if they need to share resources, it gets more complex, and the performance of shared-memory parallelism is harder to reason about.

To allow your software to be parallelized if you need more performance in the future, avoid introducing dependencies between operations if you can avoid it, such as unnecessary global variables.

> Here are two code examples using Scala. What do they do ? Can you spot the difference ? Can you tell what is the result of that difference ?
>
> >>> a.map(tup => tup._1 + tup._2).toList
>
> >>> a.par.map(tup => tup._1 + tup._2).toList

Another form of parallelism is computing in the background when possible. In an interactive or real-time system, it is good to do as little work as possible before responding to a request (see speculative execution below). The reason is twofold: first, a rapid response is better for the users, and second, the load usually varies a great deal, so there is likely to be idle processor time later in which to do background work. If you can separate some of the work from the main line of a task, then you can complete the task faster.

Lazy execution

In *lazy execution* you delay an operation until its result is absolutely needed. For instance, if initializing a printer takes a while, and many runs of the program will never print anything, then it makes sense to initialize the printer

only when the user wants to print something, instead of initializing it when the program actually says to do so (e.g., at startup). This is also true of operations that use lots of memory, such as initializing a large in-memory table to speed up a computation that users may never need.

Laziness is easier to add to existing code if the operation was already considered asynchronous: calling code does not need to be changed, and the only visible effect will be that the first execution is noticeably slower. On the other hand, if the operation was guaranteed to be synchronous before, you will either have to change all of the calling code, or accept that the program will synchronously block when the operation is first executed.

You could think of a write-back cache as a way to employ laziness for performance improvement: if an update does not need to be propagated to the backing store right away, just hold on to it and wait until later – you might not even have to do it, if the update is overwritten by a new update.

Speculative execution

Speculative execution is the exact opposite of lazy execution: perform a task before it is even asked for, and keep the result, so that when the request comes in the answer is ready. If an operation is common enough that there is good reason to believe it will be performed in the future, and there are resources available to do it either in the background or while waiting for something else, it can be a big performance win. A well-known example in hardware is that modern CPUs try to predict which way a branch will go and continue executing under that assumption, since they would otherwise have to wait for memory or cache accesses; if the prediction is wrong, the CPU backtracks.

The key to enabling speculative execution is that side effects must be controlled. Ideally, an operation has no side-effects and can be speculated freely. In practice, side-effects are common, but must be fully reversed if the speculation was incorrect, without disclosing the speculation to the external world. A good example of *not* doing that correctly is the Spectre CPU vulnerability.

Prefetching is a special form of speculative execution: you get some data ahead of time, in anticipation of that data being necessary. In other words, you speculatively fetch the data. This is often combined with caching: if you read, for example, the fist part of a file, it makes sense to bring into your buffer cache the rest of the file, because it's quite likely that you will read the rest as well. Furthermore, fetching the entire file in one go is often a lot cheaper than fetching it piece by piece.

Another way to speculate is *hedging*: if the system must perform an operation, it can call multiple implementations in parallel, assuming that one of them will be clearly faster than the others. For instance, asking a few servers in different parts of the world for the same thing is a good idea if there is a reason to believe some of them will respond significantly faster, such as because they are closer to the user's location or because they have less load. Another more esoteric example appears software verification: asking multiple solvers to solve a query, because solvers are based on heuristics and thus their relative performance can vary wildly. However, not only do the same caveats apply regarding side-effects, but hedging can decrease performance if guesses are wrong since the same work will be performed multiple times. In the case of geo-distributed servers, it may turn out that the servers could share the load if each user made one request but not when each user makes multiple ones.

Optimizing for the common case

You do not need to make your software fast in all cases, only in those that are common enough, because they will dominate overall performance. Your software "only" needs to be *correct* for all cases, not *fast* for all cases. Making the common case fast will tend to enhance performance better than optimizing the uncommon case. Plus, the common case is often simpler than the rare one, and so it is often easier to enhance. (Of course, identifying the common case is not always easy, but profiling helps a lot).

You will often hear the notion of "hot" and "cold" paths, where hot paths are frequently used and thus need to be thoroughly optimized whereas cold paths can remain relatively slow because they are rarely used. For instance,

the code to log in a user likely needs to be fast since users do not want to wait before using the service, whereas the code to change a user's password does not need to be fast.

Beware of trust when dividing paths into "hot" or "cold". If there is a way for untrusted users to make requests, then your "cold" paths could be used by malicious users to slow down the system to a crawl. For instance, it is tempting to handle malformed Web requests in a slow path that could even log the failure to disk, since these requests are not expected to happen unless a client has buggy software, but a malicious client could voluntarily overload the system with malformed requests and fill up the disk.

This principle also applies to the contexts your software might run in: you may consider some exotic cases as rare enough that they are not worth optimizing for. For instance, if your software uses the disk a lot, it could be significantly slower when run on a networked disk, i.e., a file system located on another machine. But unless you expect users to need to run your software on a networked disk, it is probably acceptable to not focus on it.

It is important to note that optimizing for the common case does not mean that the rare case never needs to be fast. For instance, the "read text out loud" feature of a word processor may be rarely used overall, but users with vision impairment need it, so if it is too slow to be useful to them, then you probably should optimize it. Similarly, using your phone with the "emergency call" feature is extremely rare, but that doesn't mean that it's OK if it has high latency.

Algorithm choices

Some algorithms are faster than others depending on inputs and environments. Sometimes an algorithm is a poor choice regardless of inputs, such as bogosort, but more often than not the choice of algorithm depends on expected inputs. For instance, the Karatsuba algorithm for multiplication is faster in theory, but in practice it only shines when inputs are hundreds of digits long, because its big-O notation hides constant factors that dominate total computation time when inputs are small.

One common kind of choice software engineers have to make is which kind of collection to use when storing a sequence of data. Using a Java `List<E>` when the code mostly performs `contains` calls can be highly inefficient, especially given how easy it is to change to a `Set<E>` when designing the code. This change is harder to make after the fact, since the rest of the code may unnecessarily start depending on ordering, for instance.

You do not have to know what inputs will look like while writing the code: you can also write code that picks which algorithm to use at runtime based on the input. This is a variant of the Factory pattern that the original "Gang of Four" book on design patterns called "Bridge".

> It is well established that quicksort may actually perform worse than insertion sort on relatively small arrays. One can use the Factory pattern to choose which algorithm to use to sort an array.
>
> ```
> def smart_sort (list):
>     if length(list) < threshold:
>         return insertion_sort(list)
>      else:
>         return quicksort(list)
> ```

Avoid copying

A common approach to defensive programming is *data copying*, for isolation purposes (copying data from one module to another ensures that neither module can modify the other's copy). Sometimes it's done to convert data from one format to another, because two modules that must interact use different representations. However, copying can take a long time.

Using immutable objects helps avoid copying, since any immutable data does not need to be copied. Even if some of the data is mutable, making parts of it immutable reduces the amount of copying involved.

If possible, use the same data format throughout the system, for instance by adopting the format used by one of your dependencies if it is already good enough for your purposes.

The following code is a common pattern in constructors:

```
private List<String> items;
public MyObject(List<String> items) {
    // defensive copying!
    // this will allocate a new array
    // and copy every single reference
    this.items = new ArrayList<>(items);
}
```

If you use immutable data instead, such as Guava's ImmutableList, you can do the following instead:

```
private ImmutableList<String> items;
public MyObject(ImmutableList<String> items) {
    // no need for defense!
    // the outside world can't attack you
    this.items = items;
}
```

Low-level optimizations

If you designed your system with performance in mind but it is still too slow, it's time to try low-level optimizations. Alternatively, if your system's design has room for improvement but you only need to make it a little faster, some low-level optimizations can make it fast enough at less cost than a redesign. In any case, while it is good to know about these tips, maintainability should always take precedence if you don't have a clear reason to optimize. Profile your code first to find the bottlenecks, then optimize those (there is no benefit to optimizing code that isn't the bottleneck).

These optimizations are also heavily dependent on the software and hardware environment, and may backfire in the future. For instance, you may optimize for a particular hardware architecture quirk now, and in one year that optimization ends up being a lot slower than "unoptimized" code, because that quirk is no longer there. Or you work around a compiler limitation that is fixed in the next release. In general, you should try to abstract such perf optimizations into a platform-specific layer, which you can then modify more or less independently of the rest of your code.

Beware of myths

When it comes to low-level optimizations, you can find plenty of terrible performance advice in books or on the Internet, either because the advice is so old it has become obsolete, or because it was never good in the first place. Performance advice can get old quickly with major changes, as compilers and runtimes get smarter and faster, leaving you with complex code that is hard to maintain but not any faster than the simpler version. A lot of performance advice is also written by well-intentioned but confused people who did not fully understand or test their advice before putting it out there, for instance because they benchmarked their changes using their own simple benchmark setup instead of a proper benchmarking framework.

It is good to look up the dates of major changes in the programming language and environment you're using if you're not sure of whether some advice is still relevant. For instance, Java got its modern collections framework

and a JIT with Java 1.2 around 1998, generics with Java 1.4 around 2004, and functional-style streams and lambdas with Java 8 around 2014. Thus, if you see an old blog post from 1997 telling you that Java collections are slow because they're thread-safe, you'll know that this applies to the old collections such as `Vector` and `Hashtable` and not the new ones such as `ArrayList` and `HashMap`.

Myths about "outsmarting" compilers are unfortunately common. For instance, the idea that an `if-else` statement is slower than a `switch` statement might have been true decades ago on early compilers, but not today. Replacing your simple string concatenations by `StringBuilder` in Java is not useful either, as the compiler will do it for you if there is no loop. If you believe the compiler is doing something inefficient with your code, look at the machine code it outputs and analyze it. This is harder to do for just-in-time compilers such as HotSpot for Java or PyPy for Python than for ahead-of-time compilers such as GCC for C, especially since just-in-time compilers might produce poor output at first then optimize the code that is run frequently.

Another common target of myths is garbage collection, which is often called out as inherently slow. While garbage collection does slow down the code, modern garbage collectors employ sophisticated techniques to reduce their overhead, such as looking at the most recently allocated objects only and doing work in the background. Garbage collection usually leads to large gains in productivity, and the small fraction of code that bottlenecks the system can be optimized to reduce garbage collection overheads as we'll see below.

Memory consumption

Using lots of memory has two main consequences: there is less memory for doing other things, and the CPU spends time allocating and deallocating objects. Leaving free memory can be useful for more caching or for running more programs on a single machine. Time spent in memory management is typically considered overhead since it does not accomplish anything on its own w.r.t. the functional requirements of the software.

To use less memory, *reuse* existing objects rather than creating new ones when you don't need to. For instance, if a word processor represents each character typed by the user as a separate object with its own properties, such as font and text color, characters should not each have their own instance of a `Font` object describing their font; instead, all characters with the same font should use the same `Font` object. This is easy to do if the `Font` object is immutable. String interning in Lisp, Java, Python, Ruby, etc. is another example of such reuse. These are all instances of a design pattern called Flyweight.

One common source of memory waste is memory leaks. A program "leaks" memory if it forgets to deallocate some part of memory it is no longer using. In languages with manual memory management, such as C++, avoiding leaks requires discipline from programmers, since they must remember when to free what memory. In languages with garbage collection such as Java and Python, all unused memory is freed, but the garbage collector's definition of "unused" is "not reachable from any used object", which is not what programmers typically think of. For instance, if you use the Observer design pattern, all observers are referenced by the objects they are observing, thus they will not be freed until the observable is freed as well. This can be problematic if the observable is long-lived: if an object observes a system event such as changes in the display language, it will stay alive for the entire lifetime of the program, even if it is no longer needed. One way to avoid this is to use *weak references*: a special way to refer to an object that will not prevent it from being garbage collected. See for instance Java's `java.lang.ref` package and Python's `weakref` module.

To spend less time allocating and deallocating memory, you can use *pools*: sets of pre-allocated objects. Instead of allocating memory for a new object, borrow one from a pool; and instead of freeing the memory, return the object to the pool. This is a specialized form of memory management, and it is effective regardless of whether you are using a garbage-collected language. However, it increases the complexity of your code and the risk of bugs, since there is nothing to prevent you from using an object after you returned it to a pool, much like "use-after-free" bugs with memory deallocation. For instance, you can use `ArrayPool<T>` in C#.

Finally, a small but impactful optimization is to reserve memory if you know you will need a lot when building sequences such as lists or strings. For instance, Java's `ArrayList<E>` has a constructor that lets you specify the initial capacity; if you know you will insert a million elements, this will avoid unnecessary data copies when the

array backing the list gets full while adding items. The `StringBuilder` class is similar for strings, avoiding the cost of building intermediary concatenations that are immediately thrown away as you add more characters.

Helping the tools

Compilers and runtimes can perform impressive optimizations, but are often unnecessarily constrained by language rules that the programmer does not actually need. For instance, if a Java programmer accidentally does not mark a method as `private`, even if it should only be used internally, the compiler has to leave that method in the final output (since it's public by default), limiting optimizations such as inlining or dead code elimination.

Here are some tips in Java, which often also apply to related languages such as C#:

- If you do not need a class to be inherited from, mark it as `final` so the compiler does not need to emit code that deals with polymorphism.
- When writing loops over arrays, stick to standard loop shapes such as the "for-each" loop or a simple "`for (int n = 0; n < array.length; n++)`", as the compiler can recognize these and avoid inserting bounds checks for the array accesses.
- Only use boxed primitive objects such as `Integer` or `Character` if you need them, otherwise use the more efficient primitive types such as `int` and `char`.

We quantitatively demonstrate how Python code can be made faster with examples in this notebook, check it out! We demonstrate why you should:

- Use Python's built-in functions
- Use for-comprehensions whenever possible to build lists, instead of for loops
- Avoid calling your own Python functions too often
- Use the right data structure (list, set, dictionary, etc)
- Use specialized modules before reinventing the wheel: they're faster and you don't have to write them
- We present some advantages of JIT compilation with Numba

Helping the hardware

Modern computer hardware is complex, and while it hides this complexity behind a relatively simple abstraction of code and data, the performance it can offer is limited by internal implementation details. For instance, code whose data fits in the CPU's caches will run much faster than code whose data does not fit in them, even if the absolute difference in data size is small. In this case, it is a matter of providing data to the CPU fast enough so it does not have to wait to begin its computation.

Let's take a concrete example in the context of game programming. Let's assume that objects in our 3D world are represented with an `Entity` structure containing various fields such as position, velocity, color, name, etc., and that at each frame, we want to update the X component of the position of each entity by a constant amount. The naive implementation would be to store all of our entities in a big array and iterate over it to modify the position of each entity.

This approach is called Array of Structures (AoS), and although very simple and effective, it has a big performance problem in this simple use case. Indeed, when loading a single entity into the cache, the CPU will not only load the position attribute but also the next attributes of the same entity (velocity, color, …). A cache line is usually 64B long (on modern x86_64 CPUs) which means that depending on the size of our entity structure, a single entity cannot even fit in a single cache line. During the next iteration, as the cache is polluted with this unnecessary data, the CPU will need to issue a second load which stalls the processor for a couple hundred cycles. A better solution would be to have all of the position attributes tightly packed into memory so that a single CPU load could store more of the same attribute in one cache line, reducing the number of necessary loads.

This other approach is called Structure of Arrays (SoA). Instead of storing individual attributes, SoA stores arrays of attributes in a single structure. Now when the CPU loads the position of the first entity, the cache is also filled

with the position of the next few entities. Coming back to the cache line size, we could fit ⌊64/(3*4)⌋ = 5 position attributes in a single cache line, essentially reducing the amount of load instructions by 5. This can in some cases greatly increase the performance of the application. However, this only works if only the position attribute is accessed in the loop. If the loop also requires the velocity attribute to determine the next entity position, then this approach will be less efficient than AoS. This method goes particularly well with SIMD extensions which are special sets of instructions that can perform a single operation on multiple pieces of data at once. This is however, out of the scope of this course but this video is a good introduction to their capabilities.

AoS entity class	SoA entity class

`class Entity {`
`  Vec3 position;`
`  Vec3 velocity;`
`  Vec4 color;`
`  String name;`
`  ...`
`}`
`Entity[] entities;`
	`class Entities {`
	`  Vec3[] positions;`
	`  Vec3[] velocities;`
	`  Vec4[] colors;`
	`  String[] names;`
	`  ...`
	`}`
	`Entities entities;`

This is an example of a data locality optimization. A general rule that can be applied to achieve better performance when it comes to loading data from memory is to keep related data grouped. Again, the CPU, although very fast, can only work if provided with data and the CPU cache is a low level form of optimization that, if used correctly, can greatly improve the performance of the code.

Ethics of benchmarking

The goal of benchmarking is to measure performance, but as we have discussed previously, benchmarks are not magical tools with perfect accuracy. Unsurprisingly, this opens the door for the manipulation of benchmarks, allowing an individual or organization to misrepresent the performance of their system (software or otherwise). There are many famous examples of corporations manipulating benchmarks or coming up with biased benchmarks. For example, the infamous "dieselgate" scandal, where Volkswagen was implicated, involved their diesel engines detecting when they were being benchmarked, and lowering their emissions (below their real emission rate) to meet EU emission standards. An example in software is when Intel and AMD clashed over the benchmarking tool SYSmark, due to the fact that the software Intel had used in their benchmarking tests had been optimised for performance only on Intel microprocessors. Another example is earlier versions of OpenOffice, which would load their code at OS boot time in order to make their startup time appear shorter. All of these unethical practices are made possible due to the fact that benchmarks are often *backward-looking* (i.e. they address issues that have already been encountered) and each benchmark *only measures one specific thing*, which means that looking at several benchmarking measures is necessary. Needless to say, you should steer clear of any benchmark manipulation or circumvention. In order to make sure that your benchmarking is fair and ethical, here are a few guidelines to follow:

1. Report all the measures that are relevant to your program, and not only those pertaining to performance. For example, if your program has security requirements, you cannot skip those requirements and then only report performance benchmarks. Remember that there are a wide variety of requirements that *could* be (but of course *should never be*) sacrificed for better performance on benchmarks, such as scalability, power consumption, availability, etc.
2. Related to the above, be sure to report your program's *range* of performance. Do not only report the mean or the best-case performance, as your system also has a worst-case performance, and users do not like nasty surprises like a web server performing much worse than the benchmark.
3. This goes without saying, but refrain from fooling the benchmarking tool you use. The OpenOffice performance hack is a prime example of using a blind spot of the benchmarking tool: the tool cannot tell if

the code of the program has already (needlessly) been loaded into memory. In “dieselgate”, VW used the fact that the benchmarking process could be detected by the vehicle, and exploited that.

4. If developing programs for a client with specific use cases and under specific conditions, report your program’s performance in those (and similar) scenarios and conditions, not in a way that is optimized for the benchmark you are using. For example, benchmark on similar hardware/software, not on hardware/software that is in some way optimized for your code (the Intel-AMD dispute comes to mind).
5. Closely related to the two guidelines above, do not optimize your code *for* the benchmarking tool, as many such optimizations can worsen performance under real conditions.

For more pitfalls that you should avoid, you can check out the “Challenges” section of the Wikipedia page on benchmarking, which goes into further detail on how many companies - purposefully or otherwise.