

The SOA Magazine

Feature Article



Software Pipelines Theory: Understanding and Applying Concurrent Processing

by Cory Isaacson

Published: October 5, 2007 (SOA Magazine Issue XI: October 2007, Copyright © 2007)

[Download this article as a PDF document.](#)

Abstract: The concept of software pipelines was developed as a methodology to enable service-oriented business solutions to implement concurrency while maintaining order of execution priorities and simplicity of application development. In order to effectively apply this methodology, it is helpful to have a solid understanding of its underlying principles. This article focuses on software pipelines law and rules and discusses the challenges and opportunities when they are applied to services-based business applications.

Introduction

This is the third in a series of articles that explores software pipelines. The first article provided an overview that included a discussion of some of the performance challenges facing application development teams especially when faced with employing highly reusable services. That article noted how multi-core or multi-threaded architectures could be leveraged via concurrent processing.

In the second article we presented two examples that demonstrate how software pipelines actually work within the context of SOA. The first involved a hypothetical network of banking automated teller machines (ATMs) and the second was an actual case study of a Customer Relationship Management (CRM) outsourcing firm.

This next article presents the theory behind software pipelines by focusing on a set of "law and rules" derived from the principles of fluid dynamics and a simplified subset of principles from "Amdahl's Law". These govern and guide the development of successful applications that utilize pipelines architecture and technology in support of SOA.

This article also discusses the challenges and opportunities presented by software pipelines when applied to business applications. There are three basic questions that will need to be answered in the development of any pipelines application:

- How do pipelines work best to accelerate service-oriented application performance?
- What are the keys to predicting and optimizing pipelines performance in support of agnostic services?
- What are the best ways to minimize the constraints and limitations of pipelines?

Pipelines Law

There is a famous adage that describes a fundamental limitation in computing performance, and it is as old as information technology itself: "All processors wait at the same speed." While it is most often used to describe underlying hardware limitations of I/O and other components that surround and interconnect with the CPU, it is highly applicable to software design and development and more important than ever in today's high-performance business applications.

Many IT organizations today will admit that they average only 25 percent utilization of their existing CPU capacity. Published studies show that even this is optimistic with the industry average of 15 percent. This leaves the other 75 to 85 percent theoretically available for up to 4+ times improvement in performance and throughput from existing hardware resources alone. Add the concept of multi-core CPUs to the equation (with two, eight or even 32 CPU "cores" per physical chip) and we can see even more unused capacity. If today's business applications are only utilizing a fraction of their current available computing power, how can we expect to capitalize on these expanded capabilities?

A key goal of service-oriented business application developers everywhere is to obtain maximum throughput in a given system, especially when working with a lot of shared services. To meet that goal, developers must increase the utilization of available processors in a way that is optimized for a given set of application problems or requirements. When processors are not busy (i.e., waiting for other resources to perform), they are not productive and, predictably, performance bottleneck results. The goal of software pipelines, and of the application of Pipelines Law, is to minimize wait time between interconnected components in an application system, thus helping to maximize processor utilization. By understanding Pipelines Law and its impact on a system, we can analyze, optimize and predict the performance of a given software system using this approach.

Many readers will be familiar with Amdahl's Law, a mathematical formula used to predict the theoretical maximum speedup using multiple processors. Although it has been in use since the mid-1960s to predict system performance (and limitations) in parallel systems, most applications of Amdahl's principles have been applied to either the "embarrassingly parallel" problem, or low-level chip and operating system architectures. Pipelines Law, on the other hand, is specifically designed to simplify the analysis of business transaction applications in order to maximize throughput. Pipelines Law is a simplified derivation of concepts similar to those of Amdahl's Law, combined with principles derived from the fundamentals of fluid dynamics.

Fluid dynamics is the engineering discipline concerned with fluids moving through a physical "pipe" system. Astonishingly, the parallels between fluid dynamics and software systems are very direct and easy to understand. The analogy to fluid dynamics helps make the principles of software pipelines easier to understand. We use the analogy in this article to affirm some basic formulas that can allow developers to predict and optimize a software pipelines system.

For example, in fluid dynamics the first and most important rule is: *Inflow equals outflow*

While this is easily seen, it is nonetheless the first and most important consideration when designing a pipeline system. Given a pipe of a certain size, you can never output more fluid on the downstream end than you fed into the pipe to begin with. Therefore, the inflow volume and rate determine the maximum possible outflow from a given pipe system (Figure 1).

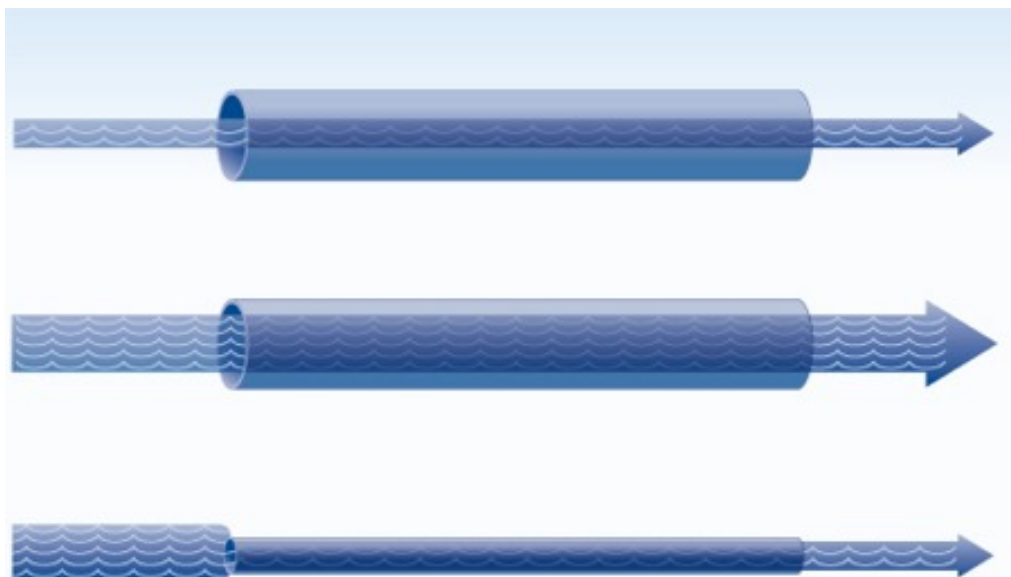


Figure 1: *Inflow equals outflow.*

This fluid dynamics rule translates into software pipelines Law as: *Input equals output* When considering a computing system as a flow of transactions, this rule shows us that we can never process (output) more transactions than the available input supply. In practice, input and output transaction rates will always be the same. However, we also know that our potential output rate will never be greater than our *available* input rate. The input rate of transactions is the "raw material" we have available to work with. Thus, the first consideration in any concurrent processing system is to evaluate the available input transaction rate, as this represents the maximum number of transactions that can be processed.

While it's true that a single input transaction can generate multiple output transactions, the law still applies. You can't process more than the available number of transactions. There must be an adequate supply to work with, and we can immediately apply the law by asking one question when we're first determining the potential usefulness of a concurrent processing system: "Are there enough transactions available to justify concurrent processing in the first place?"

Aspects of Pipeline Law

One of the most important concepts to understand in a pipelines system is the existence of limiters to flow. If an engineer can observe and predict what inhibits the flow of a given system, then the system can be optimized to ensure that the design is adequate to meet expected performance goals.

Now let's consider some important corollaries to Pipelines Law that can help illustrate pipeline behavior. The first and most obvious corollary is: *Friction and restriction limit the flow*

Again, this is easy to see. Any restriction in a fluid pipe limits the overall flow through the system, restricting and reducing both inflow and outflow. We can see this as a "crimp" in the pipe, a bend in the pipe, or a pipe of reduced diameter as shown in Figure 2.

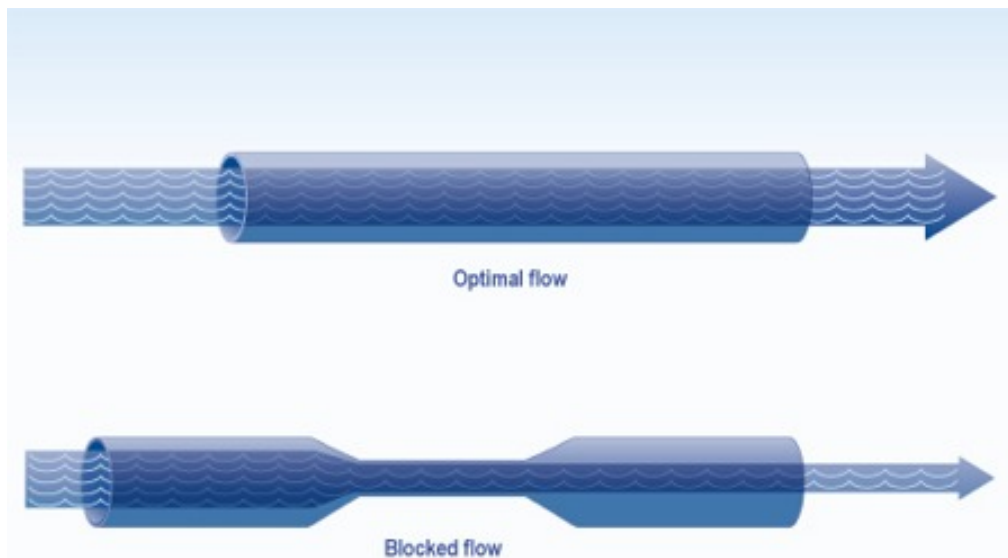


Figure 2: Restrictions to flow can limit overall throughput.

This translates to software pipelines very directly. If a single component in the middle of a system cannot process fast enough to keep up with the input flow, the processing rate of the entire system will be reduced accordingly. In other words, the processing rate is always limited by the slowest component in the system.

When a single component cannot keep up, transactions will "back up" on the input side. If that goes on long enough, it can result in a system crash due to exhaustion of memory or disk space. Therefore, it is very important to evaluate the performance of each component in a pipelines application and ensure that components are balanced to provide optimum flow through the system. Without this evaluation, the optimization of one component can create a bottleneck

in another – "downstream" part of the system.

The next corollary relates to limited flow as follows: *Restriction of the outflow reduces the overall flow*

Limiting output flow is another common way that the flow through a fluid pipeline system can be restricted. If anything restricts the output side of a fluid pipeline, the entire flow is adversely affected. Consider any blockage on the output, such as some unwanted debris, whether partial or complete. Another example is a reservoir at the end of the pipe system that overflows, thus backing up the entire system and effectively stopping the flow in the entire system (Figure 3). In other words, if anything restricts the output side of a fluid pipeline, the entire flow is adversely affected.

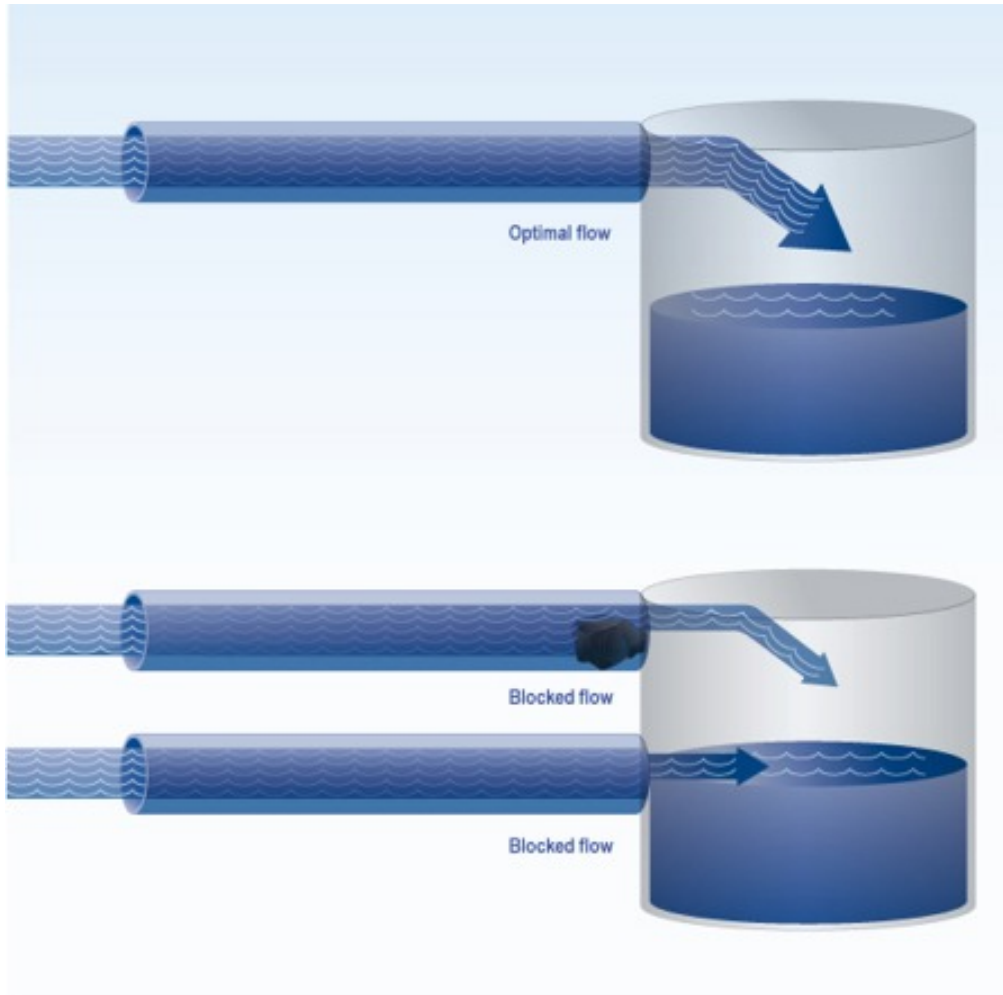


Figure 3: Blockage on the output side will restrict the flow.

The same thing is also true of a software system. Every developer has seen the effect of some centralized service that is external to the application and can't handle the load. For example, a mainframe system that is the final repository for processed transactions can be too slow to accommodate the input from many sources. Another common example is a centralized relational database that cannot keep up with transaction rates. An extreme case is where a database runs out of disk space or becomes locked, and simply stops receiving transactions.

In each of these cases, the processing rate of the entire system is either hampered or stopped completely. When this occurs, transactions "back-up," users wait and, in some cases, the entire application crashes.

Pipelines Rules

In the following sections, we explore pipeline rules and provide some very simple mathematical formulas based on these rules.

Rule 1: Input Equals Output

This is of course Pipelines Law itself, and all of the points made earlier that apply to Pipelines Law, apply to Rule 1 as

well. The success of a pipelines system requires that:

- An adequate supply of input transactions is available, justifying the use of pipelines in the first place.
- The slowest component or process in the system is identified and optimized, because it will govern the overall throughput and performance of the system.
- Any external system or application that cannot handle the load is identified and optimized, because it will present a bottleneck similar to that of a slow component or process.

So, if we have an adequate supply of input transactions, our next step is to identify each component in the system and analyze its performance characteristics in order to predict and remove bottlenecks.

The formula to express Pipelines Law is: $InputRate = OutputRate$

This formula shows that the InputRate and the OutputRate will always be the same, regardless of how many transactions there are to process. In other words, a pipelines system (or any software system for that matter) cannot accept more transactions (InputRate) than it can process (OutputRate). A more useful way of stating this is: AvailableInputRate = PotentialOutputRate.

For example, if the AvailableInputRate is equal to 10 transactions-per-second (TPS), the system can never process or output more than 10 TPS, regardless of how effective the processing actually is.

If you view this from the processing side, and assume an AvailableInputRate of 1,000 TPS, it is then easy to determine the PotentialOutputRate. If your process can handle a load of 1,000 TPS or better, then the PotentialOutputRate will also be 1,000 TPS. But what if your process can only handle 500 TPS? It's easy to see that there will be a "back up" of queued or lost transactions to the tune of 500 TPS in the system – definitely far from ideal. *Rule 2: The capacity (transaction rate) of any "downstream" component or process must be greater than or equal to the input rate supplied by any "upstream" process or component. When this is not the case, the component or process must be optimized, or a pipeline distributor must be used to support concurrent processing to meet the load.*

This rule is the key to ensuring the success of any pipelines system in meeting maximum performance and delivering on established service level agreements and business requirements. You can analyze every single point in a pipelines system with this rule. Potential and actual bottlenecks can be identified as can the need for a pipeline distributor, which enables concurrent processing for increased capacity.

The formula to represent this rule is: $InputRate \text{ must be } \leq ProcessRate$

In other words, the "downstream" ProcessRate for any component or process must be able to accommodate the InputRate supplied by any "upstream" component or process. When this is not the case, we need to increase the ProcessRate through the implementation of multiple pipelines or other optimizations. Consider the simple flow of components shown in Figure 4.

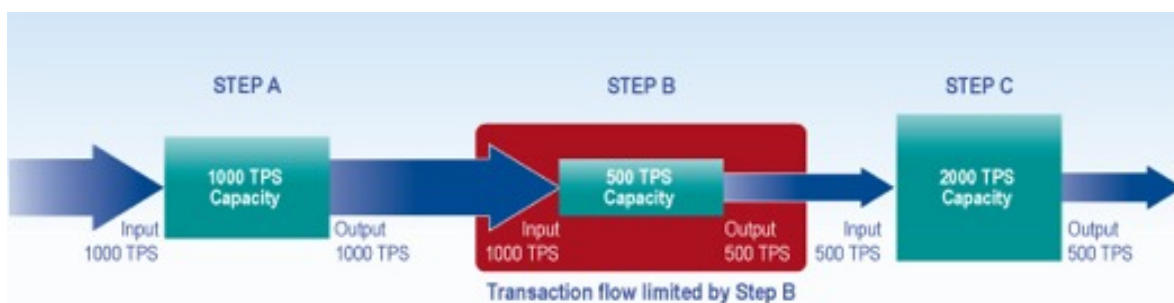


Figure 4: Step A (1,000 TPS), Step B (500 TPS Capacity), Step C (2,000 TPS).

In this case, Step B is the bottleneck that limits transaction flow. The result will be a back-up of transactions queuing up in front of Step B and limited flow of transactions into Step C such that Step C is then under-utilized. Furthermore, if there is no buffer for transactions to queue up in front of Step B, transactions could be lost (an unacceptable consequence for mission-critical applications such as banking).

To balance the transaction flow requires the application of pipelines distribution or another form of optimization for Step B that increases its throughput. Otherwise the entire flow will be limited to the 500 TPS capacity of Step B. However, if it is safe to distribute Step B, a developer or operator can simply create two pipelines for Step B to double its capacity. This will resolve the problem and balance the flow as shown in Figure 5.

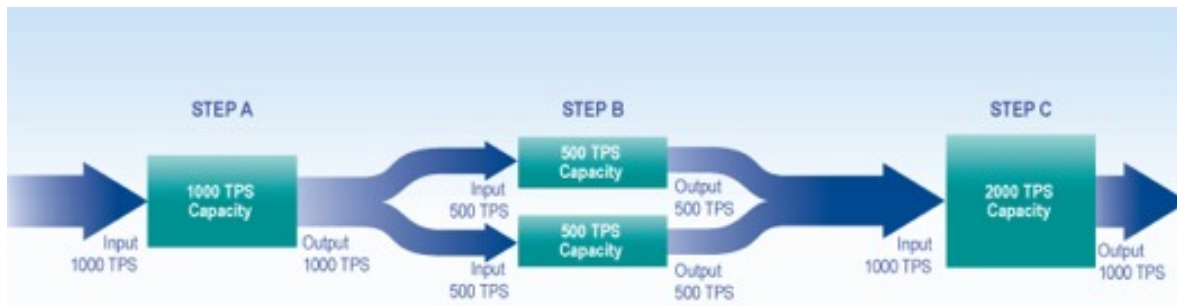


Figure 5: Software pipelines can be used to add capacity and balance process flow.

Adding a second Software Pipeline to Step B may not be enough to double its capacity if there are insufficient hardware resources to handle the increased volume. With two pipelines and adequate hardware, Step B can take advantage of concurrent processing to double its capacity (Figure 6).

However, there is one more missing link for effective concurrent processing. As shown in Figure 6, the addition of a pipeline distributor manages the dispersal of transactions across the software pipelines in Step B.

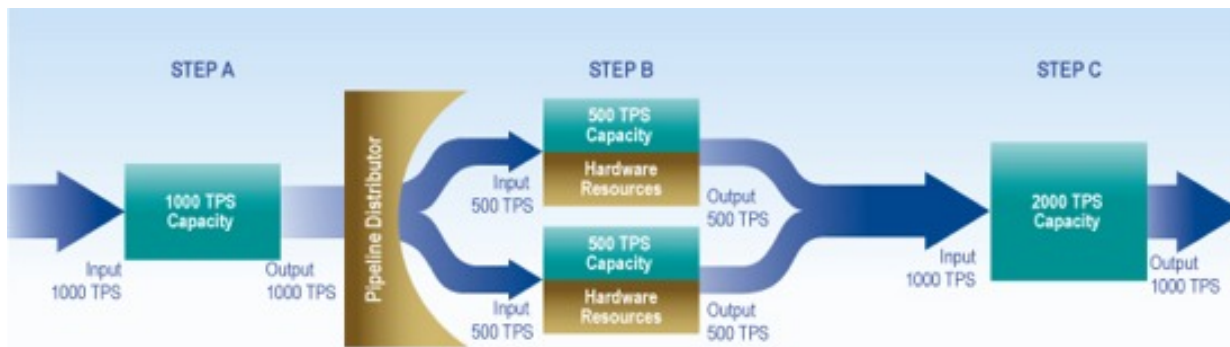


Figure 6: A pipeline distributor enables software pipelines to take advantage of concurrent processing across distributed hardware resources.

A pipeline distributor performs content-based routing on input messages, distributing the load to individual pipelines for effective concurrent processing and execution of transactions. It provides a large measure of control over the flow of transactions and supports key business application requirements such as first-in-first-out (FIFO) ordering.

The pipeline distributor is the key to allowing pipelines to take advantage of concurrent processing across multiple hardware systems, or multiple processor cores within a single hardware system. Each pipeline can be assigned to specific hardware resources so that it can execute independently and concurrently with other pipelines.

This distributor then sorts transactions into the pipelines where they can be processed independently of transactions in other pipelines. You can then add hardware resources to a given pipeline if the hardware itself becomes the bottleneck. Because the pipelines are independent of each other, adding more hardware resources along with more pipelines enables both linear or near-linear scalability.

Taken to an extreme, it's easy to see that adding a large number of software pipelines, with enough hardware resources under the control of a single pipeline distributor, will eventually create a bottleneck at the pipeline distributor itself.

This leads us to Rule 3.

Rule 3: The processing rate of the pipeline distributor must be far greater than the downstream processing rate.

There are several things to consider when implementing a pipeline distributor. The first and foremost is that the work of

distribution always adds overhead to the system. For pipelines distribution to be effective, the pipeline distributor must perform its work far faster than the actual process that it is supplying.

This can be expressed with the formula: $DistributorRate \gg ProcessRate$

In other words, the DistributorRate must be far greater than the "downstream" ProcessRate. If it's not, the pipeline distributor will create a new bottleneck in itself, thereby defeating the entire purpose of the software pipelines architecture.

An example is shown in Figure 7, where the distributor must have at least a 2,000 TPS throughput capacity in order to avoid becoming a bottleneck.

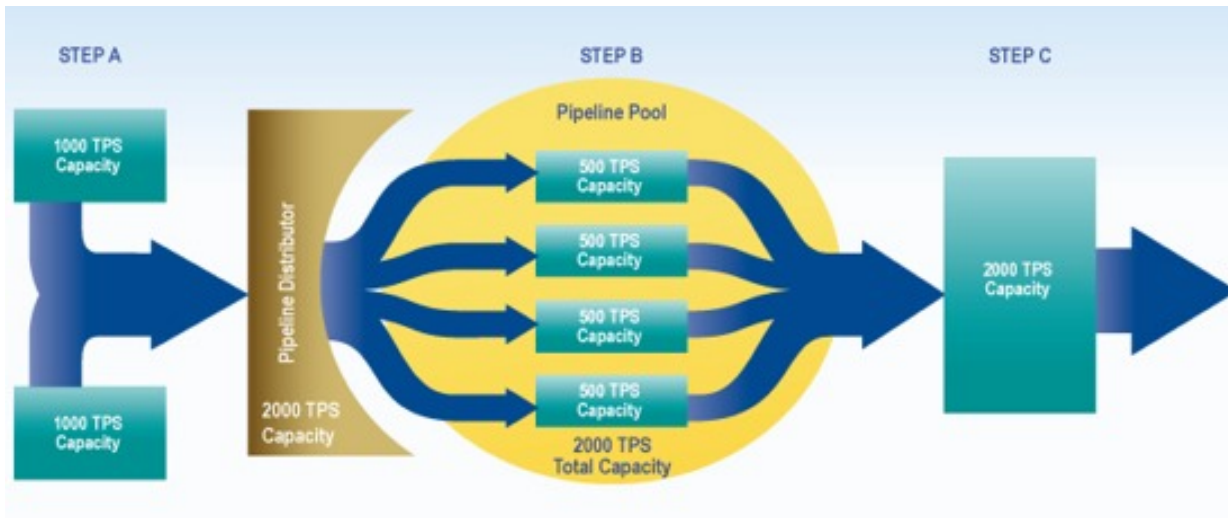


Figure 7: Pipeline distributor throughput must at least match downstream throughput.

If the distributor in the diagram can process 2,000 TPS, and feed four pipelines for Step B that can each handle 500 TPS, then everything will work fine.

But what would happen if the distributor uses very complex logic for determining the correct pipeline and routing of transactions, and could only route transactions at the rate of 1,000 TPS? As Figure 8 shows, this would simply create a bottleneck at the pipeline distributor. So, there would be no benefit from implementing pipelines, and the pipeline distributor would be unnecessarily wasting valuable computing resources.

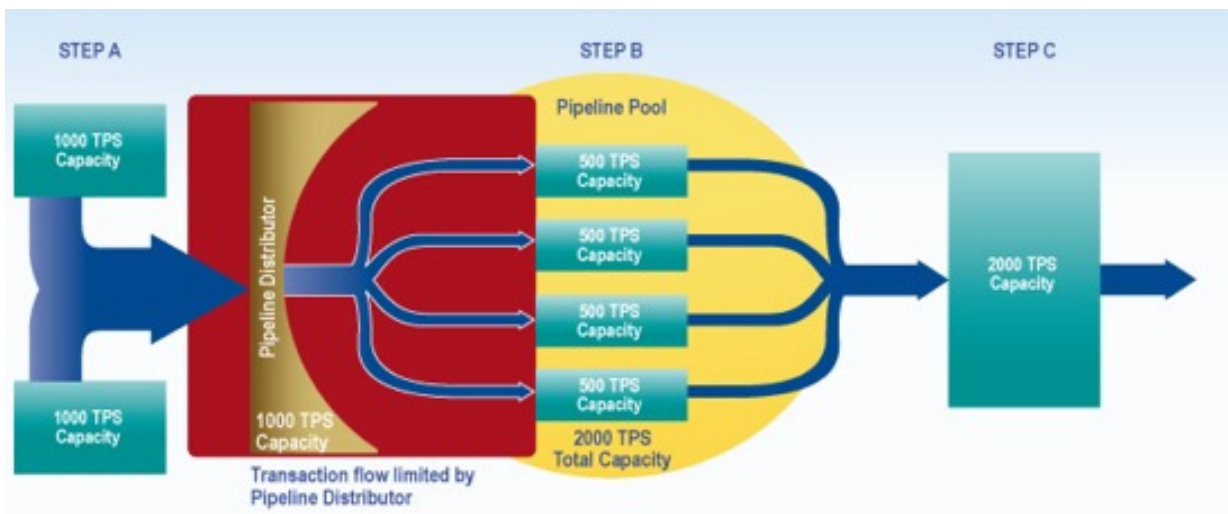


Figure 8: A low throughput pipeline distributor will create a bottleneck.

Another problem can come about if you try to distribute too many pipelines. Assume again that the distributor can process 2,000 TPS. But instead of feeding four pipelines as in the optimal flow of Figure 7, this time it routes the transactions to eight pipelines as shown in Figure 9. In this case, the distributor would only be able to feed 250 TPS to each instance of Step B running on a given pipeline, yet Step B pipelines can each handle 500 TPS. Again, we are

wasting resources and our system is not optimized.

Figure 9 shows eight pipelines in Step B for a total transaction throughput capacity of 4,000 TPS. Yet the pipeline distributor feeding these pipelines has a capacity of only 2,000 TPS. It can therefore only send enough transactions for half of the capacity of Step B.

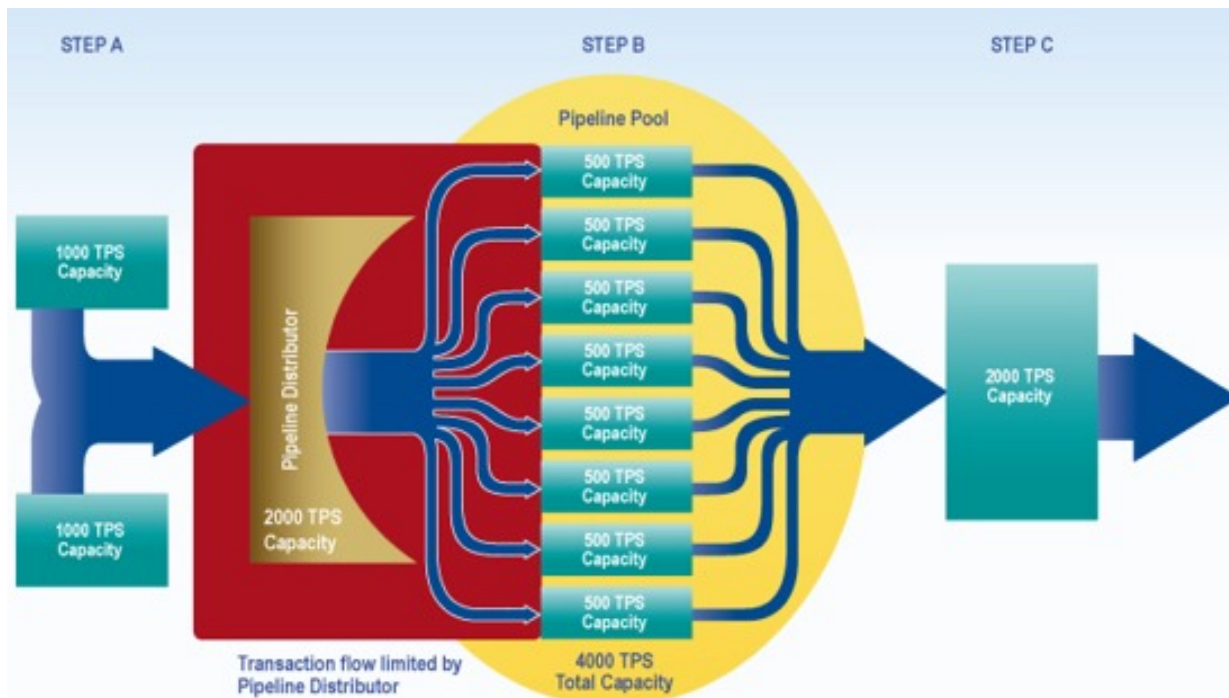


Figure 9: Adding more pipelines than the pipeline distributor can handle will also waste resources.

These examples lead us to a formula that can be used to determine the right number of pipelines for a given part of a system: $NumberOfPipelines = DistributorTPS / ProcessTPS$

The formula simply states that the theoretical maximum number of downstream execution pipelines (or subdivisions of a given process) is the ratio of the distribution rate (DistributorTPS) and the downstream processing rate (ProcessTPS). The formula shows that the theoretical ideal number of effective pipelines (NumberOfPipelines) is determined by this ratio. Of course, real-life systems demand some margin for error and it is thus typical to "over design" to some degree. You may want to plan to increase the NumberOfPipelines by 10-20 percent to allow for this margin of error and help ensure adequate flow in the system. Used in this way, the formula can provide an excellent guide for properly sizing a system and determining the optimum application of a pipeline distributor in a pipelines system.

For a very simple example, let's assume that a given distributor can evaluate and route transactions at the rate of 1,000 TPS, and the downstream process runs at the rate of 100 TPS. In this case, the maximum number of pipelines would be 10, expressed as: $10 = 1000 / 100$

This pipelined system should be able to perform up to 10 times as fast as a comparable system that does not utilize pipelines technology. More specifically, the system should be able to process a throughput of 10 times the number of transactions in a given amount of time.

The formula is designed to show that downstream execution pipelines must be "supplied" at a rate comparable to the rate they can process, or "consume" transactions. Otherwise the execution pipeline processors will simply wait, and will be unproductive resources in the system.

Therefore, in the worst possible scenario, if the Pipelines Distributor rate requires more time to evaluate and route a transaction than a single execution of the actual transaction process, there is no benefit to concurrent processing. In fact, quite to the contrary, the distributor would simply introduce unnecessary overhead into the process. For a service-oriented application to be successfully "pipelined", the design goal must be to minimize the latency of the distributor, while delegating as much work as possible to downstream execution pipelines.

Consider also that pipelines can be distributed at multiple levels in a given system as required. Rule 3 must be applied at each level of the pipeline distribution hierarchy.

Conclusion

Pipelines Law and the three derived Pipelines Rules provide a very simple, yet effective method for evaluating a software pipelines system. To be sure, there are many other limiting factors that need to be taken into account, such as network latency, high service reuse, contention, etc. However, these rules can be applied to the analysis and prediction of performance for just about any service-oriented business application that requires the scalability offered by concurrent processing.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

