

The SOA Magazine

Feature Article



High Performance SOA with Software Pipelines

by Cory Isaacson

Published: March 1, 2007 (SOA Magazine Issue V: March 2007, Copyright © 2007)

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

Abstract: Many organizations have adopted SOA development models to deliver flexible and agile application components, but frequently ended up having to trade off performance and scalability in order to achieve these benefits. As service-oriented applications continue to grow in size and complexity, performance has become a foremost concern.

Various optimization techniques exist to address runtime service performance concerns. This article explores these established design options and contrasts them with software pipelines, a new technology developed to accommodate service usage demands without compromising the flexibility and agility that is so important to achieving strategic SOA goals.

Introduction

As organizations continue to accelerate growth, business applications are consistently required to run faster, support more users, and process more transactions. With limited budgets and no easy way to expand data center capacity, IT executives are faced with the increasingly difficult challenge of squeezing more performance and greater utilization out of existing systems - all while delivering greater flexibility so that systems can be adapted in response to rapidly changing business needs.

These challenges are driving today's IT executives to:

- Find faster and more efficient ways to process growing volumes of data without increasing capital expenditures.
- Build flexibility into business systems so that new services can be efficiently provisioned in order to capitalize on new market opportunities or comply with emerging regulatory requirements and industry mandates.
- Quickly adjust allocated resources for IT services in order to respond to unpredictable spikes in user demand.
- Drive down IT costs by improving system utilization and IT efficiency.

In today's information-based economy, application software performance can literally mean the difference between success and failure. Because employees and customers are so dependent on automated systems, poor application performance can directly impact an organization's bottom line. Solutions with erratic or deficient response times can cause very real damage, including lost revenue, poor customer satisfaction, and a negative image in the marketplace. Mission critical applications (such as banking, trading, call centers, and on-demand reporting systems) that suffer from sustained, inadequate runtime performance can cripple a line business, leading to a reduced capacity for the organization as a whole to compete.

Runtime performance has always been a primary concern for the professional developer. Throughout IT's history there have been numerous techniques, approaches, and even architectures offered to specifically address performance and scalability. While the promise of SOA surpasses that of previous IT architectural models in terms of flexibility,

adaptability, and strategic gain, it is known for imposing performance demands. Therefore, with service-oriented application development now on the forefront, performance optimization is receiving more attention than ever.

Some common factors include the following:

- Due to the increased emphasis on reuse and agnostic design, services are subject to unprecedented levels of concurrent usage demands.
- The very notion of loosely coupled services infers a messaging-centric communications framework, meaning that applications must now handle not only traditional processing logic, but also message transmission, validation, interpretation and generation.
- As the use of service-oriented concepts proliferates, messaging volumes are expected to explode, adding tremendous load that will push the limits of IT infrastructures.

SOA-related performance issues that organizations will experience during the next 12-24 months are expected to be similar in nature to the growing pains of previous software architectures when they were “new.” Looking back over the past two decades, the shake-out period associated with each new major paradigm shift in software development introduced significant performance during the first years of wide adoption.

Furthermore, the learning curve imposed by these paradigms often resulted in deployed applications that failed to meet performance expectations and, in many cases, required newly developed applications to be prematurely retired.

The Evolution of the CPU

In the past, developers have been able to rely on rapid advances in CPU performance to compensate for the lack of software efficiency in their business applications. With CPU clock speed doubling every 18 months or so, an upgrade of the hardware environment often provided enough of a performance boost to keep up with the growing need for application throughput.

More recently, gains in CPU clock speeds have hit a plateau due to uncompromising physical factors such as power consumption, heat thresholds, and quantum mechanics. Current trends in hardware platforms have thus shifted the focus to multi-core or multi-threaded architectures.

Distributed service-oriented applications, by their nature, take advantage of multi-CPU and multi-server architectures. However, for software applications to truly leverage multi-core platforms, they must be designed and implemented with an approach that emphasizes concurrent processing. This new approach, based on a methodology called *software pipelines*, can enable businesses to achieve the benefits of concurrent processing without major redevelopment effort.

Concurrent Computing

In order to accomplish dramatic “multiples of performance,” applications must execute more than one task at a time. While this may be obvious, it is not easy to accomplish. Even service-oriented applications, which are already distributed in nature, may be designed to use a serial approach to processing business logic so that the proper order of execution is maintained. It can be difficult to decompose the business logic of the application into a series of steps, some of which can then run concurrently.

Historically, the computer science field has performed extensive research and developed many techniques to accomplish concurrent architectures. Yet, the focus of past research and development concentrates on specific areas that do not easily lend themselves to the transactional applications of today’s business systems.

Therefore, while the need for substantial performance improvements of business applications clearly exists and is becoming more pronounced due to the wide-spread transition toward SOA, existing concurrent processing techniques are either limited in applicability or too complex to incorporate. There are four primary approaches that have been developed for concurrent computing, each of which is discussed in the following sections.

Symmetric Multi-Processing (and Mechanical Solutions)

Mechanical solutions at the operating-system level have no doubt benefited many organizations to date by providing a generic one-size-fits-all approach to concurrent computing. Symmetric Multi-Processing (SMP) platforms automatically split running application tasks among multiple processors on a single physical computer, sharing memory and other

hardware resources. This approach is highly efficient and easy to implement, as the application developer needs no detailed knowledge of how the SMP divides the workload.

For SMP to be effective, however, a software application must be written using multi-threaded logic. This is a tricky task that is not generally practiced by corporate IT developers. Furthermore, the tight sharing of resources between processors is both limiting in terms of performance and problematic when applied to business application needs. Shared resources become a bottleneck at some level of scalability, as the necessary locking of resources in this type of system is not optimized for any particular application.

Therefore, a given application may scale well to eight processors, but benefit very little from applying 16 processors to the problem. In addition, resource contention (such as shared software components) can be very difficult to debug within this type of black box environment.

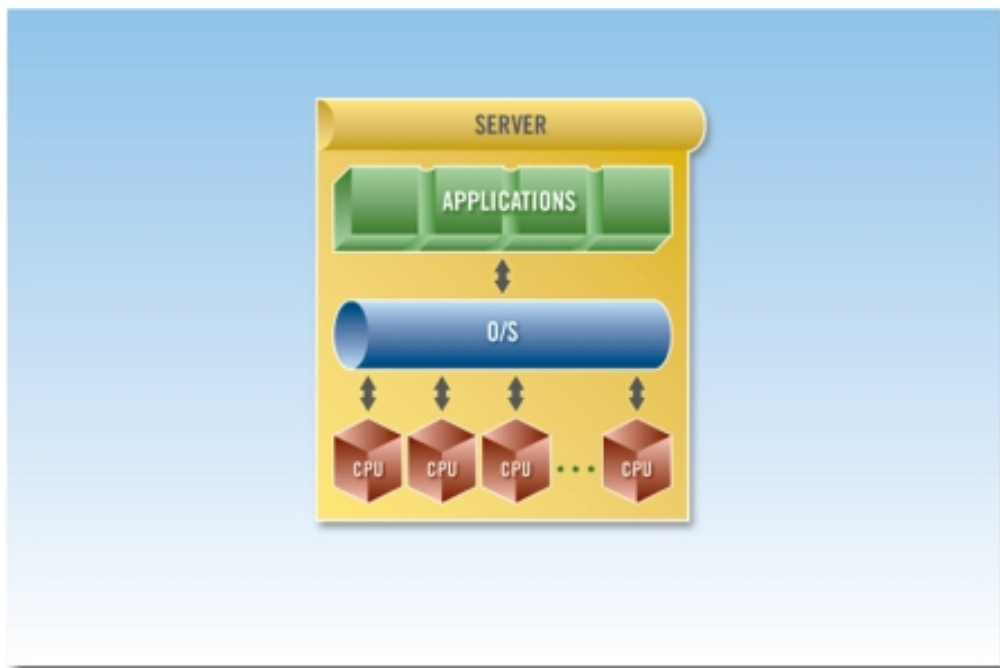


Figure 1: An SMP server operating system manages the workload distribution across multiple CPUs.

Automated Network Routing Solutions

Systems enabled with automated network routing divide application requests using some form of predetermined logic. A common approach is “round-robin” routing, where requests are evenly distributed, one after the next, among a set of physical computers that provide exactly the same application functionality. A good example and use case for this type of concurrent architecture is a Web server, wherein each Web page request is delivered to one of several available processors. While the approach can be useful, it is highly limited as the router has no concept or logic for determining the best path for a given request. Furthermore, all downstream processors perform identical processing tasks.

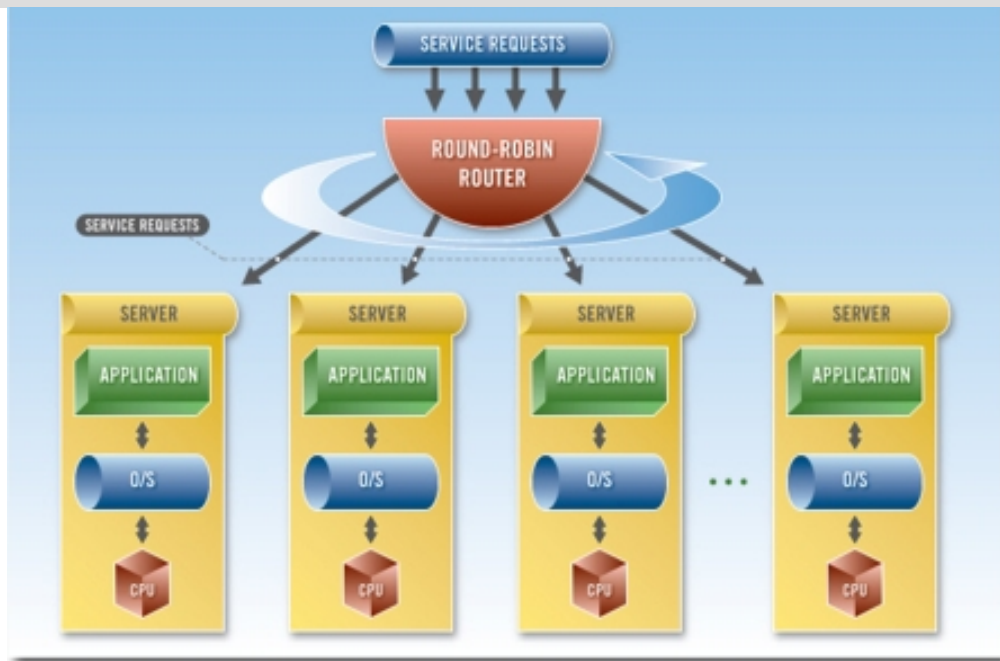


Figure 2: In a round-robin network routing approach service requests are routed to individual servers in a pool of redundant servers.

Clustering Systems

Clustering is a widely used technique that allows physically separate computers to share the workload of an application over a network. Clustering provides some capabilities for automatic concurrent processing and is also used to support fail-over and redundancy. In this scenario, redundant resources are replicated across the network, resulting in a highly inefficient approach. Because clustering techniques are automated, they must copy everything from one node in a cluster to another whenever a change in state occurs. Alternatively, they must rely on a centralized resource (such as a relational database), which can become an even more serious bottleneck.

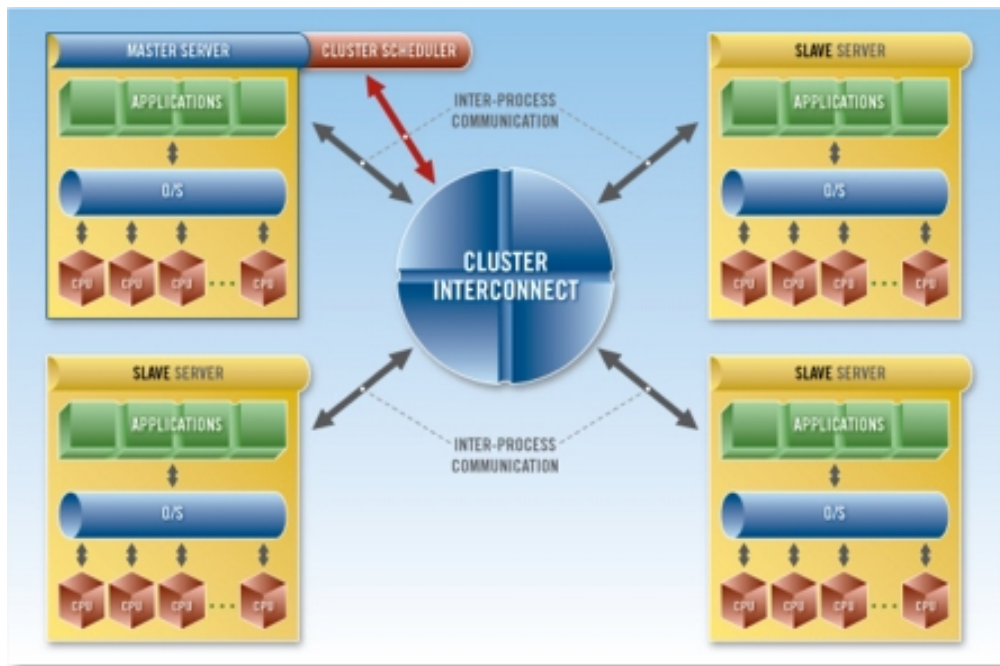


Figure 3: Within clustered systems, multiple servers share common resources over a private "cluster interconnect".

All of these techniques serve a purpose, yet each is limited when it comes to massive scalability, particularly when considering the needs of transaction-based, message-centric solutions. In essence, they can only scale mechanically and automatically to a certain level, at which point the overhead of maintaining shared or redundant resources becomes more of a burden than the resulting performance improvement.

Grid Computing

The formation of a resource grid can achieve great scalability by distributing discrete tasks across many machines in a network. In a grid computing environment, it is left to the developer to decide how best to:

1. Divide a single large task into smaller sub-tasks.
2. Utilize the grid environment to distribute the processing.
3. Reassemble the results once processing is complete.

The typical grid architecture includes a centralized task scheduler for distributing and coordinating the tasks with other computing facilities across the network. It has been shown that a grid approach can deliver far higher throughput than the automated approaches described earlier. However, this option can also place an increased burden on the developer due to the previously listed responsibilities.

Most importantly, grid computing has been modeled primarily to solve the “embarrassingly parallel” problem – long-running, computation-intensive processes commonly found in scientific or engineering applications. Typical and productive examples of grid computing applications include the modeling of fluid dynamics, the tracing of the human genome, and complex financial analytics simulations. Each of these application areas has the common characteristic of dividing a massive, long-running computation among multiple nodes, decomposing the problem into smaller, similar tasks that tend to behave in a predictable manner when considering computational resources.

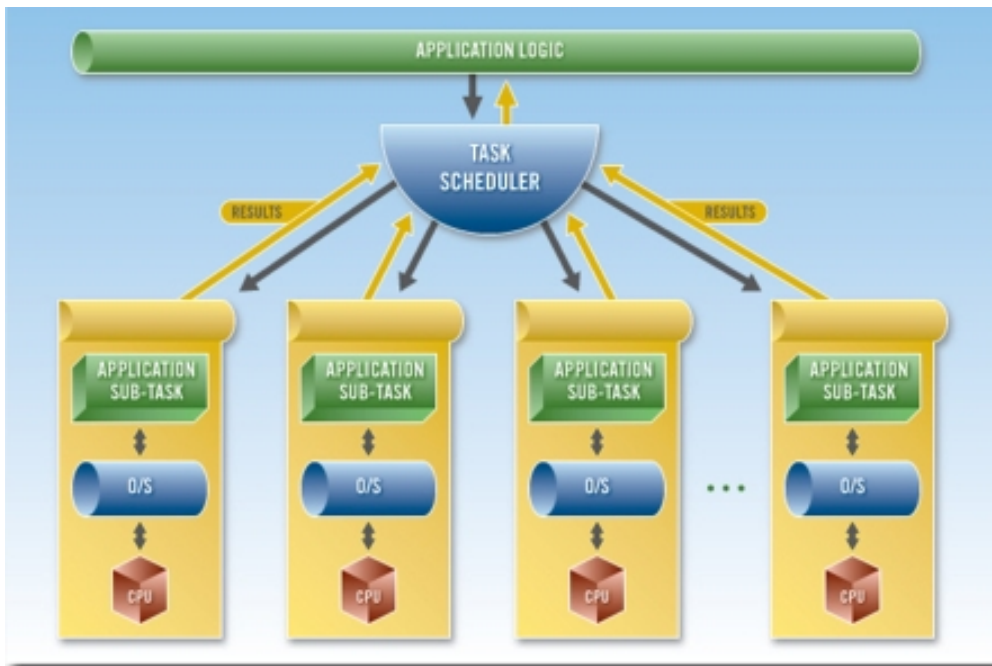


Figure 4: Within a grid computing platform applications are divided into sub-tasks that can execute independently.

Limitations of Traditional Concurrent Processing Approaches

There are three primary reasons that service-oriented business applications do not lend themselves to traditional concurrent processing techniques.

1. Order of Processing is Critical

Business logic must be performed in a specific sequence to ensure the integrity of the business process. In many cases, applications implement a “first in/first out” (FIFO) queue by waiting for each transaction to be completed before the next one in the queue is processed. For example, a billing application cannot compute the total cost of a bill before it has looked up the rates that apply to the customer and computed subtotals for each different category of services. For a mobile phone bill, the business logic would need to know the total daytime, evening, and weekend minutes before it can compute the bill total.

This order of processing is difficult to maintain in a grid computing environment. While SMP systems are designed to ensure order of execution (unless the application is written with multi-threaded logic) there can be significant performance problems when the volume of transactions reaches a critical threshold.

2. Centrally Shared Resources Create Bottlenecks

Although services are ideally designed with increased autonomy, in most environments service architectures involve the usage of a shared database or other centralized resource. In a typical concurrent processing environment such as an SMP server or a grid infrastructure, the centralized resource presents a bottleneck that limits application throughput. Resource contention eventually creates a performance problem if transaction volumes continue to increase.

3. Unpredictable Behavior and Resource Needs

Compared to a massively parallel scientific application, the average service is much less predictable in its runtime behavior and resource needs. The size and processing requirements of business workloads can vary greatly throughout the day or even within a given hour. This not only makes it more difficult to divide service logic into equally sized components in terms of processing time required, but it also means that allocation of resources must be flexible enough to dynamically respond to the resource requirements of each component.

Introducing Software Pipelines

Software pipelines introduce a new concurrent processing methodology that provides a simple way for service-oriented business applications to implement concurrency while maintaining order of execution priorities and simplicity of application development.

The software pipelines architecture supports peer-to-peer scalability and the decomposition of business processes into specific tasks. These tasks can be subsequently executed in parallel, while the overall workload can be balanced across the resources within one or more servers. It also provides a means by which developers can control the distribution and concurrent execution of various tasks or business process components.

The software pipelines architecture is designed to handle a high-volume stream of transactions, both large and small, and, thus, is ideal for mixed-workload application processing.

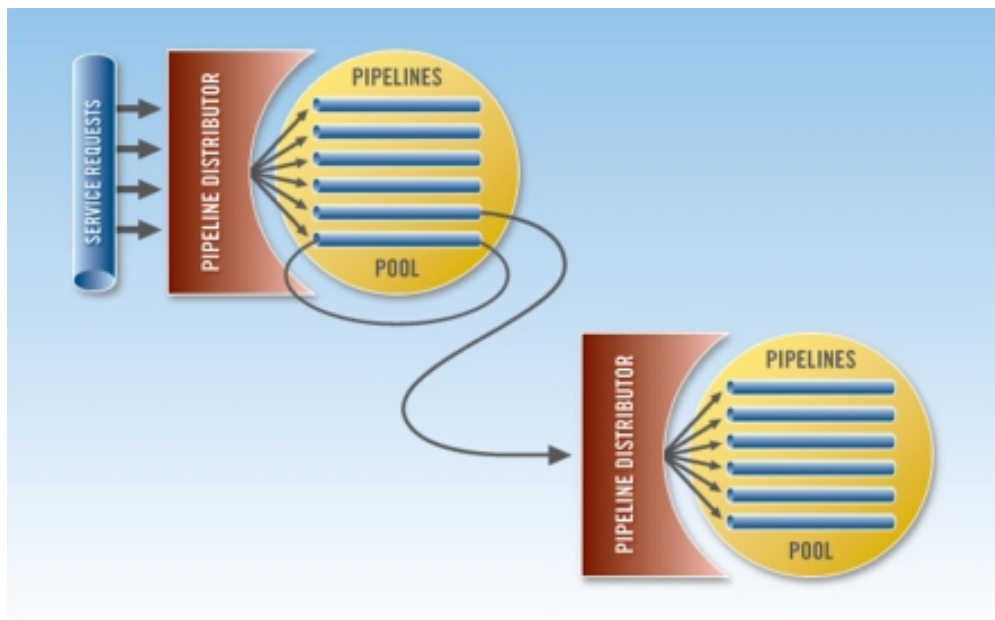


Figure 5: Software pipelines are front-ended by a pipeline distributor that routes service requests.

The fundamental architectural component is the pipeline, which is defined as: “A logical execution facility for invoking the discrete tasks of a business process in an order-controlled manner based on priority, order of message input, or both.”

Within a service-oriented solution, software pipelines can be used to group transactions or business logic for which order of execution or priority must be preserved. For example, each customer of a bank might be associated with a specific pipeline. The pipeline would then execute all computations and transactions that relate to the specific customer and preserve the order of execution for transactions that relate to that customer. Other customer

transactions could then be executed on different pipelines, which could process those transactions without regard for the order of the first customer's transactions.

For customer billing transactions there is no concern about whether one customer's transactions are completed before another. Different customer bills can be processed concurrently on separate pipelines while transactions that relate to a single customer bill are processed sequentially within the same pipeline.

Controlling Pipeline Flow

To implement the software pipelines approach, a *pipeline distributor* is needed for sorting service requests from the business application into appropriate pipelines and for balancing the load across multiple software pipelines. This component is co-located with a pool of pipelines, and effectively front-ends incoming service requests as shown earlier in Figure 5.

Pipeline distributors route service requests by evaluating message content and in response to *configuration rules* that can be easily modified without changing individual business services. These configuration rules can be established and modified to distribute workloads and to optimize throughput via concurrent processing according to priority and/or the order of input (FIFO).

This design approach enables scalability in two dimensions. Additional pipelines can be added under a given pipeline distributor, and when a distributor has as many pipelines as it can effectively manage, more pipeline distributors can be added. When more pipelines are added to a system, the capacity for managing additional transaction volume grows proportionally. Furthermore, workloads can also be moved between pipelines to avoid bottlenecks.

Working with Software Pipelines

One of the major advantages of the software pipelines approach is that it is simple to implement. Multi-threading an application can require that developers dissect the application into discrete components that run concurrently without disturbing the business logic. With software pipelines the business logic can remain intact and the sorting into pipelines can be based on unique identifiers (such as customer IDs), which are often already maintained as part of the business logic.

Software pipelines furthermore enable developers to:

- implement concurrency only in the performance-critical portions of the business logic
- enforce FIFO only where required
- control distribution of system resources to help maximize utilization

The business rules that are used by pipeline distributors to sort service requests can be modified to tune performance and redistribute workloads. Pools of pipelines can also be allocated to a specific hardware resource and can be moved to take advantage of new hardware resources as they are added to the computing infrastructure.

Conclusion

SOA has delivered big gains in terms of flexibility and adaptability, but has left some organizations wondering whether a service-oriented solution can meet their business application performance needs and expectations. Today's IT organizations are looking for more efficient ways to process rapidly growing volumes of data and take better advantage of new multi-core hardware platforms.

The software pipelines methodology offers a simple and easy method for business applications to exploit concurrent processing without the large development investment required to add multi-threading to a traditional monolithic business application. They can enable service-oriented applications to support higher transaction volumes within existing IT budgets and help ease the strain on data center capacity.

Finally, this approach to performance optimization results in tangible business benefits, including the reduction of capital expenditures, increased business requirements fulfillment, and more cost effective development processes – all of which are in alignment with the key strategic goals of service-oriented computing.

