Choice of Design Techniques for Scaling Mission Critical High Volume Applications

Muralidaran Natarajan
BIT Mesra
Mumbai, India
Email: muralidaransk@gmail.com

Nandlal L. Sarda
IIT Bombay
Bombay, India
Email: nls@cse.iitb.ac.in

Sharad C. Srivastava
BIT Mesra
Ranchi, India
Email: sharad.scs@bitmesra.ac.in

Abstract**--** In today's Services business, self-servicing models with thousands of users operating "Anytime Anywhere" concurrently and generating millions of transactions have increased the performance requirements multi-fold. However, these systems are monolithic and are not flexible for business transformations and changes. To scale beyond this, many organizations have started to transform these systems to new generation of multi-core platforms. This being architecturally different, the existing applications will have to be decomposed and re-architected to run as parallel components, have near zero contention and take advantage of these new generation platforms. This paper studies the choice of design techniques with the elements of concurrency and parallelism to arrive at an optimal setup to scale the performance of high volume mission critical real time applications on modern platforms. The choices are further validated with the simulation runs conducted to narrow down the combination setup giving the best performance. The results, analysis and the recommendations aim to provide an approach to the design architects in transforming mission critical applications.

Keywords- layering; concurrency; parallelism; mutual exclusion; visibility of change.

I. INTRODUCTION

Any business today has a critical dependence upon information that is acquired, processed, transported and delivered with well-defined Quality of Service (QoS) [1]. The ultimate intention of these organizations is to attain a competitive advantages in decision-making, customer service, and to respond in a timely manner to changes in the current market place and stakeholder expectations.

Traditionally, large enterprise On Line Transaction Processing (OLTP) solutions relied on using clusters of mainframes running proprietary information systems software. Many of these applications were developed in low level/procedural languages and evolved over a period of time [2]. Originally, the applications were designed to implement specific business models and related functions to support enterprise automation in the monolithic form. The monolithic form in the single processor offered good performance benefits with very minimal inter process communication overheads in the earlier era. As the requirements expanded, the applications were modified using wrapper-like methodology to satisfy the business demands without any design change, thus ending up with sub-optimal solutions [3].

With business growth and emerging markets, the changes in the business scenario resulted in new business models. Often, the transaction patterns and the concentration/volume of each pattern turned out to be very different from the original business objectives and specifications based on which systems were designed and developed. The emerging self-servicing models and ease in customer service using pervasive new generation devices/interfaces multiplied the access points and interfaces. This resulted in an increase in security issues [4].

Over time, the flexibility of the monolithic systems considerably reduced and reached a breakpoint beyond which the system can only continue to run "AS IS" without much modification; hence, unable to handle the changing trends and demands of the emerging business. This resulted in the need for transforming these legacy OLTP systems to agile real-time data stream processing applications to assure high throughput and low latency [4].

Typically, Pre-Processing, Core Business Processing and Response Processing (R-Service, referred to as "Response Delivery Layer" in this paper) are the key components of any mission critical real time application. The "Response Delivery Layer" can be parallelized for performance using multiple senders as this stage is normally stateless. This flow involves Central Processing Unit (CPU) bound activities, memory bound activities and Input/output (I/O) bound activities.

This paper analyses and provides insight to the application developers on different architecture constructs to scale the performance of only the "Response Delivery Layer" in mission critical real time high performance application systems towards meeting the QoS expectations of the end customer. Section II describes some of the key architecture approaches and implementation challenges in the transformation. Section III describes the design techniques that can be used in the process and the implementation challenges. Section IV presents the performance study for the implementation alternatives discussed in previous sections, details and results analysis of the simulation runs. Section V provides conclusion and future work.

II. ARCHITECTURE APPROACHES AND IMPLEMENATION CHALLENGES

In this era of multi CPU and multi-core, large computing power is available at the disposal of the developer. It is imperative that the developer looks at various architecture approaches and design alternatives to exploit the computing power available in delivering high throughput and low latency in service oriented mission critical real time systems that are in use.

This paper examines the following architecture approaches and provides design and implementation recommendations to the developer in transformation of mission critical real time business systems to take benefit of the computing power available in the new generation hardware: (i) Layered architecture, (ii) Concurrency and Parallelism and (iii) Use of higher computing power of multi-core technology [5][6]. In addition, these architecture approaches are proposed to be studied with the possible design models, design techniques, interaction models across tasks and key aspects of implementation in the new multi-core environment available.

A. Layered architecture approach for high throughput

The immediate previous generation mission critical real time high volume application systems were architected for clustered main frames, and were designed with procedural languages and tools as building blocks. These systems which support the core business functions of the organization became large monolithic pieces, with more and more interfaces getting added to support business growth and changes without architectural redesign, rendering these systems sub-optimal. Typical systems like railways and airline reservations, retail banking, etc. have the following tasks: (i) receiving the inputs, validations and pre-processing, (ii) business processing and (iii) response and delivery of output as part of a large monolithic module. Although, this makes the coding and development simpler, it poses limitations; it makes the modules tightly coupled and rigid. Any addition/change, be it functional or technical, results in changing the core of the system; thereby, making the development, testing and release process complex and lengthy [7]. Further, it also increases the risk to the organization. Localizing the errors for a faster response to guarantee the Service Level Agreement (SLA) becomes a challenging task. To introduce self-servicing and straight through processing, as the interface handling is not standardized, the handshaking and processing are not clearly demarcated from the core system functionality. This introduces limitations in handling throughput and latency thereby constraining the business expansion and customer values. This necessitates the implementation of a multi-tiered and layered architecture to introduce flexibility and scalability [8][9].

In a layered architecture, the monolithic application is split into manageable and logical functional layers [5]. In each of the layers, the processing is decomposed into various tasks, and each task is connected using a suitable Inter Process Communication (IPC) mechanism. This IPC mechanism, which enables the upstream/downstream communication between the tasks, is referred to as "connector" in this paper. Given that a complex process can be divided into a series of sequential tasks, this approach can provide increased performance under the following three types of computational scenarios: (i) In a multi-step business process, the modularity helps in initiating the next task before the final completion of the previous task. As soon as the core transaction is complete, the information can be passed on to the next task in a modular design and internal operations like logging and safe store commit can continue off the critical path, thereby isolating the performance of "critical transaction path" from other internal operations [10]. (ii) if the processing can be done in parallel for multiple data sets until a certain stage in the processing path, the sequencing of the process steps assist in creating multiple instances of these tasks. Having multiple instances of a task minimizes the idle time of CPU and offers faster response. Running multiple instances also offers fault-tolerance and load balancing [11][12]. If an instance fails due to exceptions, another instance can take care of further processing. If there are multiple processors, the instances can be distributed across those processors [10]. (iii) In a multi-step business process using the modular design, any change can be localized at a task level, making the development and deployment process simple with moderate risk.

In order to introduce concurrency and parallelism, the modular design uses various models, such as: data parallel model, task graph model, master-slave model, pipeline model and hybrid model [13]. In the data stream application, pipeline and master-slave models are commonly used [14]. These models do not decrease the time for processing a data element, but are aimed at increasing the performance of the system while processing a stream of datasets. A modular system requires more resources as one task cannot use the resources of the previous task as compared to the monolithic case. Also, the task to task communication introduces some latency. It is therefore necessary to balance the number of tasks carefully for optimal throughput and latency.

B. Concurrency and parallelism for high performance

Once a monolithic process is decomposed into multi-step business sub-processes or tasks, the next goal is to achieve concurrency and parallelism in execution of these sub-processes or tasks to improve the overall end to end throughput and latency of the process chain. In data stream based service oriented systems, where thousands of retail users accessing the self-service facilities are simultaneously connected and transacting, implementation of concurrency and parallelism plays a crucial role in the architecture construct and choice of design techniques to achieve the desired throughput and latency demands of the changing business models and scenarios.

1) Concurrency for high performance

In software, concurrency is two or more tasks happening in overlapping time period. They may be interacting with each other and also they could contend on access to resources. The contended resource may be a database, file, socket or even a location in memory [15]. When a business function can be segregated into logical tasks and they can possibly interact and make progress in overlapping time periods, they can be deployed in concurrency mode to maximize the performance.

The interaction models for concurrent tasks are message passing connectors and shared memory, and the mode of interaction is either synchronous or asynchronous communication [16][17]. The elements of concurrency are threads, events, notification routines etc [5][15]. The challenges in concurrent execution are handling mutual exclusion not resulting in deadlocks, serialization where needed and visibility of change across concurrent tasks.

2) *Parallelism for multi-fold performance scaling*

While concurrency enables maximum throughput on a single processor, parallelism increases the throughput multi-fold by executing tasks simultaneously on a multi-core/multiprocessor hardware that supports multiple process execution at a time [15]. Over four decades, the performance of the hardware processors has been continuously increasing, and the software applications have been constantly adopting upgrades to take benefit and satisfy the business demands. With the single core becoming more complex, power consumption was becoming a bottleneck, which has resulted in a new era of multiple simple cores on a chip. Due to the complete change in hardware architecture, applications designed for single core processors will have to be re-designed to take advantage of the multi-core processors capabilities in delivering high throughput and low latency.

The two key factors that decide the benefits of parallelism are the (i) length of the critical path of the business transaction and (ii) need based optimal access to the shared resources [18]. If the length of the critical path is nearly the same as total work, the parallelism is of no significance, however powerful the parallel hardware is. Similarly, the critical section that manages the shared resources could be a bottleneck in parallelism as it serializes the access [19][20]. The methods to control this are by minimizing the length of the critical section and/or by limiting the number of threads/tasks entering the critical section at a time.

While more threads introduced for concurrency minimize the idle time and maximize the CPU usage, the optimal number of threads needs to be carefully architected depending on the number of cores available and the contention bottlenecks. The Operating System (OS) schedules the threads to the CPU in a round-robin time-priority scheduling. Every schedule change involves a context switch and takes a few hundreds of CPU cycles [21]. The transaction/data access patterns may not be consistent and could vary widely under various scenarios. A good implementation is the one that has bounded contention, not completely driven by end user transaction/data access patterns and limited context switching. In high volume systems, it is recommended to use thread pools to have bounded number of threads for latency predictability rather than dynamically creating threads [18][22]. CPU affinity may have to be introduced to avoid threads being constantly shuffled across cores or processors.

III. DESIGN TECHNIQUES AND IMPLEMENTATION CHALLENGES

This section of the paper discusses the implementation challenges and techniques to address those challenges in interaction models such as message passing queues and shared memory; concurrency elements such as threads, events, notification routines; key aspects like mutual exclusion and visibility of changes; the use of layering, concurrency and parallelism [23]. Major mission critical systems like banking, trading, telecom, aviation and e-commerce applications that are reliable and flexible are based on message driven and self-servicing business models. These systems support low response time and high throughput due to their ability to process multiple messages concurrently using component based layers.

A. *Techniques for high performance in message passing*

Queues and shared memory are used as connectors to exchange information across concurrent tasks. However, in high performance systems queues are preferred over shared memory [16][17]. The message communication can be point-to-point messaging queues or publish subscriber model. The following sub sections describe the techniques for boosting the performance of the message passing systems [24]. Queues are introduced for point-to-point communication. Senders publish or write the messages to the queue and the receivers receive or read the messages from the queue. Producers and consumers can be dynamically added and deleted allowing the message queue to enlarge and collapse as and when needed. The producer can send a message irrespective of the consumer being up or down, and the consumer can read when it comes up. The order in which the messages are consumed depends on the priority, expiration time and the processing capacity of consumers.

A simple point-to-point messaging consists of a producer and consumer sending and receiving messages through a queue. It is possible to increase the throughput by having multiple queues. The producer needs to maintain the list of queues in an array, and select the next queue to be used for sending from the array. The algorithm to choose the right queue for sending is crucial.

Throughput of a messaging system can also be increased by introducing multiple consumers to a queue under specific conditions. Although multiple consumers can receive from a queue, a message from the queue can be consumed by only one consumer. Adding multiple consumers to a queue increases the overall throughput of the messaging system, provided the order of processing is not significant. When multiple consumers access a queue, the load balancing among them takes into account the consumer's capacity and the message processing rate. A more complex messaging system would consist of multiple queues each having multiple producers and consumers connected.

There are situations in which the multiple consumers implementation cannot be used to increase the throughput, due to high resource consumption and the need to preserve the order of the messages. It then forces the configuration of one consumer per queue. In such cases, the publish/subscribe model is recommended. The publish/subscribe model normally uses topics and is generally used to broadcast information. This model is significantly faster than point-to-point, as it is a push technology. The queues are to be replaced with topics and needs to ensure that more than one subscriber is not listening to the same topic; you get the implementation similar to point-to-point but much faster because of less overheads in such implementation.

While queues and topics are persisted and made durable within the messaging system, there is an impact on performance. Persistence can be selectively used in a messaging system, depending on the need and criticality [24]. For e.g., if you are running batch processing application and reading the input from a file/database and sending through a messaging component, in case of system failure, you can restart the processing from the last transaction and overwrite the prior processing and hence persistence is not essential. The non-persistence is to be implemented carefully to boost the performance.

B. Contention management for high performance

Real-time transaction processing with large volumes demands scalable access to rapidly changing data, ensuring consistency across millions of transactions and thousands of users. The multi-threaded processes in these applications for concurrency need to be re-designed to exploit the high degree of hardware parallelism introduced in the recent multi-core architecture towards boosting the performance. The increasing number of parallel processes increases the contention, and poses new challenges. The cost of mutual exclusion locks and degrades the performance of parallel applications significantly. The shared resource could be any object ranging from a simple block of memory to a set of objects, and instructions protected through a critical section.

In high performance applications, the design of contention management plays a significant role. It is therefore important to make the right choice of mutual exclusion methodology. Blocking based approaches introduce overheads on the critical path due to context switching and is further complicated by scheduling decisions resulting in priority inversion. Non-blocking (spinning) approaches consume significant processor resources. Non-blocking is attractive because there is no idle time as soon as the shared resource becomes available to the waiting process. However, non-blocking produces a type of priority inversion by hindering the lock holder from running and releasing the lock, thus limiting scalability [18][25][26].

But the blocking holds the waiting thread from running and allows other threads to use the complete CPU resources. Once the shared resource is available, the waiting thread has to be scheduled again. This introduces bottlenecks in the critical path, at least by 2x. Blocking is robust but makes very bad outliers [26]. These outliers (around 0.1%) are normally 5-10 times the normal latency. In high throughput systems, any delay not only affects that event but thousands of subsequent transactions. Any resource creation and freeing has a cost and therefore, to avoid jitters, the design needs to accommodate creation of synchronization objects, reuse/recycle and free them as a housekeeping job.

The length of the critical section again is a factor with the increase in number of threads and contention. The length of the critical section, and the duration of the held locks need to be optimal to achieve the desired result under heavy loads [27].The cost of entering and leaving the critical section has to be taken into account if there are too many very short critical sections. In case of long critical sections, the scheduling and priority inversions will impact the performance. As critical sections have a direct impact on performance, the coding should be done in language and libraries with native support and not a wrapped up piece which is meant for open systems integration. There has to be careful balance in implementing the blocking approach, non-blocking approach and the length of critical section. As discussed above, with the increase in contention due to concurrent and parallel processing there is a need to move towards fine-grained synchronization instead of coarse-grained synchronization [28][29].

C. Memory management for high performance

The technology advances have transformed the hardware and software platforms for applications from single processor to multiple cores and from single process to multi-process/multi-threaded to increase the throughput. However, the memory managers have not changed on par. The memory management is a key factor in designing the architecture of a system for high performance.

As the processors/cores are added, the application gradually degrades due to heap contention. Memory managers must therefore offer less space overheads, limited defragmentation and speed. Memory allocation and freeing has a performance cost in terms of CPU cycles as they involve a switch between the user mode and kernel mode and will impact the predictability of latency and response time. In critical systems, unbounded memory consumption poses the risk of crash with too many small, frequent memory allocations and freeing. Every memory allocation has the overhead of metadata and in too many small allocations, the overhead will be high. The effective solution is one having bounded memory by allocating a large chunk and managing the application requirements within the chunk [30].

IV. PERFORMANCE STUDY FOR THE DESIGN ALTERNATIVES

A. Performance study - Overview of the "Response Service" module taken up for study

In trying to address the transformation challenges described in the previous sections, design options emerged through a rigorous mix of the various approaches, combining the design models, design techniques and addressing the implementation challenges that we saw as being deterrents in restricting the scalability and performance of such real-time service oriented systems. The approach also focuses on ensuring the minimization of contention. In pipelined systems with multiple numbers of readers and writers on queues, in our performance study, it is ensured that any data should be owned by only one thread for write access; thereby, eliminating write contention completely.

This section describes the details of performance studies conducted to arrive at a suitable design approach and implementation nuances of design techniques to overcome the limitations, to meet the QOS objectives to support high throughput and low latency. The typical flow of a mission critical real time processing application is as follows; **Pre-Process** takes care of routine housekeeping, filtering, enrichment, master updates and just in time dynamic validation if any. Besides, it handles the stateless validations that may be carried out before reaching the business tier in parallel. **Core Business Process** does the actual business transaction processing taking into account the external events and update/processing of the incoming transactions and essential I/O that may be needed for recovery in a real-time mission critical application. **Response Service** (R-Service) is the process responsible for sending the output to multiple receivers and interfaces over network (referred to as “Response delivery Layer” in this paper). The same can be parallelised for performance using multiple senders as this stage is normally stateless. In this particular exercise, the design alternatives are tested only in the response or outbound path. However, the other two components are retained to simulate end to end behaviour of such real-time event processing systems.

In high performance systems, wherein the output of the business process is to be sent real-time to the end-users, the receiving of the input is independent of sending the response. The two activities proceed independently. To handle heavy loads and huge number of connections, clusters of “Processing Machines” and “Connection Machines” are deployed. However, if there is huge backlog at the dispatching of response, it creates heavy back pressure on all the previous stages. The “R-Service” process that is after the “Core Business Process” stage in the end to end processing path is considered for validating the design alternatives towards improving the throughput and response time.

In this, the “R-Service” process which is in the outbound path is used to deliver the processed information to the end consumer and there are no specific SLAs in the maintaining sequence of the messages across the end users. Hence, this service can be looked at as a potential candidate for exploiting the abundant computing power using (i) modular design approach, (ii) concurrency and parallelism in introducing multiple publishers and (iii) dedicating processing power by way of assigning individual cores to the publishers in picking up the packets from their respective queues and forwarding to the end consumers.

In the proposed design approach, “R-Service” process is broken into the following 3 tasks (a) “R-Processor”, i.e., response processor, (b) “R-Sender”, i.e., response sender and (c) “R-Gateway”, i.e., final end user delivery unit. In addition, the design techniques and the limitations are examined and factored to maximize the throughput and reduce the latency. The following design techniques and implementation variations are used in the performance study (i) Queues vs. shared memory grid (ii) Bounded queues in the form of ring-buffers vs. unbounded shared memory grid (iii) Pre allocated memory buffers by pre-filling upfront for the bounded queues for faster access and to overcome issues of fragmentation (iv) With and without CPU core binding at task level to make dedicated processing power available and thereby improving the predictability of the task by minimizing the jitter.

B. Performance study – Implementation alternatives

1) Design alternative 1: Pipelining with blocking shared memory IPC connectors

In this setup, the approach of pipelining is used as design model (Figure 1). The “R-Service” process is split into 3 different tasks, the “R-Processor” and the “R-Sender” on the business processing machine itself and the “R-Gateway” on a different hardware. In addition, the following are the design techniques considered for the implementation. (i) The communication between the sub process/task “R-Processor” (Writer) and “R-Sender” (Reader) is through a shared memory grid. (ii) For synchronization of access to shared grid, writer and reader use locks, i.e., blocking mode of mutual exclusion is used for the read write operation. (iii) Every “R-Sender” is single threaded, which services a set of business users connected to one “R-Gateway” (One to one). (iv) Communication between the sub-process/task “R-Sender” to “R-Gateway” is using topics of any messaging application.

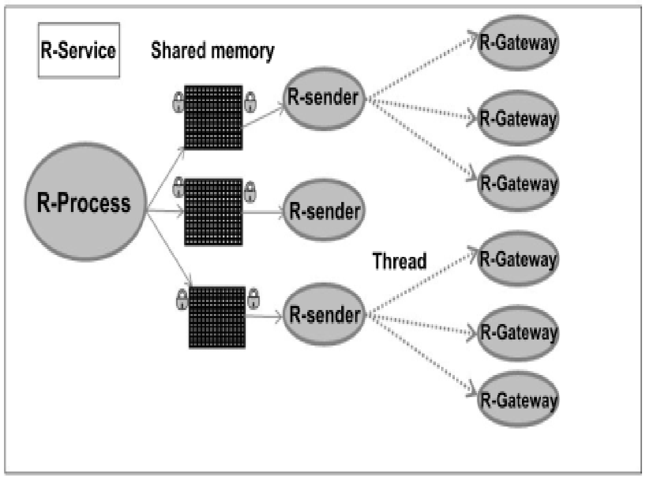

Figure 1. Pipelining with Blocking Shared Memory IPC connectors
.

2) Design alternative 2: Using bounded circular FIFO queues for IPC

In this setup shared memory grid is replaced by queues and concurrency is introduced in “R-Sender” (Figure 2). In addition, the following are the design techniques considered for the implementation along with variation in number of “R-Senders” and CPU core binding. (i) The communication between the “R-Processor” (Writer) and “R-Sender” (Reader) is through bounded circular First in First Out (FIFO) queues. (ii) Bulk read is used for boosting the performance in case of any lag in the “R-Sender”. (iii) The queue elements are pre-filled and initialized to avoid fragmentation and faster access to increase the throughput (iv) Each “R-Sender” is connected through its own queue to the “R-Processor” and hence there is a single reader and writer for each queue. The Lock-free non-blocking design technique is used in the implementation. (v) Concurrency is introduced in “R-Sender” using multi-threaded in sending data to multiple “R-Gateways”. (One to Many) (vi) “R-Sender” publishes data using different topics. Each “R-Gateway” listens only to a specific topic. (vii) Parallelism is used by varying the number of “R-Senders” and the CPU core binding to arrive at the optimal implementation mix for improved performance.

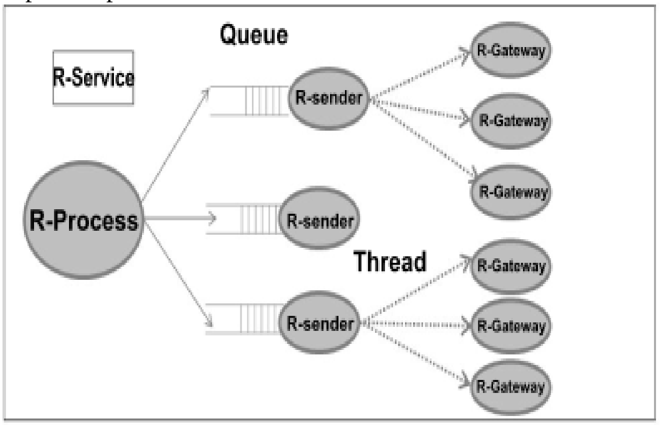

Figure 2. Bounded Circular FIFO Queues for IPC.

C. *Implementation alternatives - Simulation studies*

With the above setups, multiple runs were carried out to ascertain the behaviour of the design alternatives and impact of the design techniques by varying the number of “R-Sender” with and without processor affinity enabled.

The study runs were executed with a 2 CPU Xeon Intel processor with 8 Core each running linux6.x operating system. The CPU 0 was used for the critical path services and the CPU 1 was used for all other ancillary services. Within CPU 0, Core 0 was assigned to OS; Cores 1-3 were used for “Receiver” and “Business Processor” and 4-7 were used for “R-Service” components.

“Blocking with Shared Memory (run 1)”: This run was carried out using the setup in alternative-1 with 20 “R-Senders” and 20 “R-Gateways”. This is to record the latency response behaviour of the service using shared memory for message exchange and in blocking mode. This is used as baseline to compare the performance of non-blocking implementation variations.

Non-blocking with Queues (run 2): This run was carried out using setup in alternative-2 with 20 “R-Senders” and 20 “R-Gateways”. This is to record the latency response behaviour of the service in the non-blocking mode using FIFO circular-buffer. Queues were used instead of shared memory for message passing.

Non-blocking with Queues and Resources binding (run 3): This run was carried out using setup in alternative-2 with 6 “R-Senders” and 3-4 “R-Gateways”. In addition dedicated CPU core was allocated for each “R-Sender” and the corresponding threads to study the concurrency and parallelism. This is to record the latency response behaviour of the system in the non-blocking mode and with dedicated computing resources.

Non-blocking with Queues and Resource binding and ”R-senders” variation (runs 4-7): These runs were carried out using setup in alternative-2 with 3, 4, 2 and 1 “R-Senders” and each having multiple “R-Gateways”. In addition, dedicated core was allocated for each “R-Sender” and the corresponding threads to study the impact of concurrency and parallelism. This experiment is to record the latency response behaviour of the system in the non-blocking mode and with dedicated computing resources and variation in the number of “R-Senders” to arrive at a sweet spot combination for optimal performance and throughput.

1) *Performance study - Results and analysis*

The percentage of packets processed for each of the latency buckets is shown in Table 1.

TABLE 1. PERCENTAGE OF PACKETS PROCESSED FOR EACH OF THE LATENCY BUCKETS.

	Response Senders						
	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**
Latency	**Baseline**	**20**	**6**	**4**	**3**	**2**	**1**
21-30	**0.131**	**24.117**	**47.103**	**86.954**	**83.946**	**81.684**	**77.379**
31-40	**6.193**	**63.822**	**86.802**	**96.045**	**94.470**	**93.937**	**88.011**
41-50	**16.171**	**83.320**	**94.035**	**98.500**	**97.902**	**97.629**	**92.370**
51-60	**34.237**	**89.286**	**97.432**	**99.419**	**99.189**	**98.987**	**94.884**
61-70	**62.610**	**93.289**	**98.915**	**99.741**	**99.685**	**99.581**	**96.559**
71-80	**82.029**	**95.674**	**99.523**	**99.862**	**99.858**	**99.807**	**97.746**
81-90	**91.530**	**97.400**	**99.754**	**99.920**	**99.926**	**99.893**	**98.594**
91-100	**95.296**	**98.554**	**99.843**	**99.945**	**99.956**	**99.932**	**99.211**
101-110	**97.091**	**99.275**	**99.889**	**99.955**	**99.969**	**99.951**	**99.638**
111-120	**98.109**	**99.664**	**99.913**	**99.962**	**99.975**	**99.960**	**99.881**
121-130	**98.732**						
131-140	**99.110**						

• It is observed that in non-blocking mode (run2) performance has improved over blocking mode (run1).

a) 90 percentile order latency reduced by 30µs. Latency moved from 90µs to 60µs.

b) 99 percentile order latency reduced by 65µs. Latency moved from 140µs to 110µs.

• Variation in number of “R-Senders” with dedicated computing resources shows that optimal number of “R-Senders” minimizes the contention and improves the performance further.

• In case of 6 “R-Senders” (run 3) over the baseline, i.e., blocking (run 1), the following are the observations:

a) 90 percentile order latency reduced by 45µs. Latency moved from 90µs to 45µs, 50% improvement.

b) 99 percentile order latency reduced by 65µs. Latency moved from 140µs to 75µs, 46% improvement.

• In case of 3 “R-Senders” (run 5) over the baseline, i.e., blocking (run 1), the following are the observations:

a) 90 percentile order latency reduced by 55µs. Latency moved from 90µs to 35µs, 61% improvement.

b) 99 percentile order latency reduced by 80µs. Latency moved from 140µs to 60µs, 57% improvement.

However, further reduction in “R-Senders” to 2 and 1 in runs 6&7, the performance degrades compared to the above two runs, indicating the number of parallel processes is not sufficient to handle the load. Hence it is important to arrive at the number that gives the optimal performance. The results are under random rate of input arrival to simulate real life conditions. Under uniform input arrival rates, the performance is expected to be better. The architecture along with design components are to be fine-tuned to specific problem situation for optimal implementation. Specific care is to be taken to keep the critical path short and concurrent tasks with minimal or zero contention in the critical path.

V. CONCLUSION AND FUTURE WORK

This paper discusses the use of layering with concurrency and parallelism for high throughput and performance in a multi-core hardware platform. Also, the research work covers the choice of design techniques for interaction models and contention management in the architecture approaches. In addition, memory management strategy is suggested, taking into consideration the volume of data handled by these systems. Based on the above approach, simulation results are compared for the implementation alternatives, viz., blocking with shared memory, non-blocking with queues, non-blocking with queues and resources binding, and all these parameters with variation in number of R-sender processes. In summary, the study is aimed at providing a ready reckoner to the developer community the technical concept, implementation alternatives and choice of implementation based on simulation results in the transformation exercise.

In the end to end processing path, besides the “Response Delivery Layer”, the other layers such as Pre-Process, Core Business Process, I/O in the critical path and in-memory database are being taken as future work items.

ACKNOWLEDGEMENTS

We acknowledge the support of National Stock Exchange of India Limited for the lab facilities for the research work.

REFERENCS

[1] D. A. Menasce, V. A. F. Almeida, L. W. Dowdy, and L. Dowdy, Performance by design: computer capacity planning by example, P. H. Professional, Ed. Prentice Hall Professional, 2004, pp. 11–33.

[2] L. Wu, H. Sahraoui, and P. Valtchev, “Coping with legacy system migration complexity,” in Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCSŠ05), June 2005, pp. 600-609.

[3] L. Zhang, I. Al-Azzoni, and D. G. Down, “Pele-an MVA-based performance evaluation methodology for computer system migration,” Journal of Software Evolution and Process, 2013, pp. 1-29.

[4] EDS, “Financial services legacy modernization,” EDS View Point Paper, Tech. Rep., 2007.

[5] C. Michiels, M. Snoeck, W. Lemahieu, F. Goethals, and G. Dedene, A Layered Architecture Sustaining Model Driven and even driven software development, Springer, Ed. 5th International Andrei Ershov Memorial Conference, PSI 2003, July 2003, vol. 34, no. 4, pp. 58-65.

[6] H. R. Simpson, “Layered architecre(s): Principles and practice in concurrent and distributed systems,” in Engineering of Computer-Based Systems, 1997. Proceedings, International Conference and Workshop, March 1997, pp. 339–350.

[7] R. Rice, “Regression testing in large, complex and undocumented legacy systems,” Technology Transfer, Tech. Rep., June 2012.

[8] R. Peacock, “Distributed architecture technologies,” IT PRO, June 2000.

[9] IBM, “Building multi-tier scenarios for web sphere enterprise applications,” IBM Redbook, Tech. Rep., 2003.

[10] B. Wilkinson and M. Allen, Parallel Programming Techniques and Applications using Networked Workstations and Parallel Computers. O Prentice Hall, 2006, pp. 140-158.

[11] E. M. Karanikolaou and M. P. Bekakos, “A load balancing fault-tolerant algorithm for heterogeneous cluster environments,” Neural, Parallel & Scientific Computations, March 2009, pp. 31-46.

[12] Y. Shi and G. D. V. Albada, “Efficient and reliable execution of legacy code exposed as services,” in Computational Science - ICCS 2007: 7th International Conference, May 2007, pp. 390-397.

[13] A. Grama, A. Gupta, G. Karipis, and V. Kumar, An Introduction to Parallel Computing: Design and Analysis of Algorithms. Pearson, 2009, pp. 139-142.

[14] S. Kleiman, D. Shah, and B. Smaalders, Programming with threads. Prentice Hall, 1996.

[15] C. Breshears, The Art of Concurrency: A Thread Monkey’s Guide to Writing Parallel Applications, M. Loukides, Ed. O’Reilly Media Inc., 2009, pp. 1-20.

[16] T. J. LeBlanc and E. P. Markatos, “Shared memory vs. message passing in shared-memory multiprocessors,” in Fourth IEEE Symposium, December 1992, pp. 254-263.

[17] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim, “Integrating message-passing and shared-memory: Early experience,” in PPOPP ’93 Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, July 1993, pp. 54-63.

[18] F. R. Johnson, R.Stoica, A. Allamakee, and T.C. Mowry, “Decoupling contention management from scheduling,” in ASPLOS XV Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, ACM, New york, March 2010, pp. 117–128.

[19] R. Johnson, I Pandis, and A Ailamaki, “Critical sections: re-emerging scalability concerns for database storage engines,” in DaMoN ’08 Proceedings of the 4th international workshop on Data management on new hardware, June 2008, pp. 35-40.

[20] M. A. Suleman, O. Mutlu, and M.K. Qureshi, “Accelerating critical section execution with asymmetric multi-core architectures,” in ACM SIGARCH Computer, March 2009, pp. 253-264.

[21] R. Mark, “High performance messaging,” NFJS Magazine, Tech. Rep., 2011.

[22] D. Kimpe, D. Carns, P. H, Harms, K, Wozniak, J.M, Lang, S, and R. B. Ross, “AESOP: expressing concurrency in high-performance system software,” in Networking, Architecture and Storage (NAS), June 2012, pp. 303-312.

[23] A. Gupta, A. Tucker, and S. Urushibara, “The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications,” in SIGMETRICS 91 Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems, May 1991, pp. 120-132.

[24] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazières, and Frans Kaashoek, “Multiprocessor support for event-driven programs,” in USENIX 2003 Annual Technical Conference, June 2003, pp. 239-252.

[25] R. Johnson, M. Athanassoulis, R. Stoica École, and A. Ailamaki , “A new look at the roles of spinning and blocking,” in DaMoN ’09 Proceedings of the Fifth International Workshop on Data Management on New Hardware, June 2009, pp. 21-26.

[26] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, “Optimal strategies for spinning and blocking,” Journal of Parallel and Distributed Computing Volume 21, Issue 2, May 1994, pp. 246-254.

[27] D. Dig, J. Marrero, and M. D. Ernst, “How do programs become more concurrent: a story of program transformations.” in IWMSE 11Proceedings of the 4th International Workshop on Multicore Software Engineering, May 2011, pp. 43-50.

[28] A. Turon, “Reagents: expressing and composing fine-grained concurrency,” in PLDI ’12 Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, June 2012, pp. 157-168.

[29] B. Saha, A. R .A. Tabatabai, R. L. Hudson, C. C. Minh, and B.Hertzberg, “McRT-STM: a high performance software transactional memory system for a multi-core runtime,” in PPoPP ’06 Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM press, New York, March 2006, pp. 187–197.

[30] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, “Hoard: a scalable memory allocator for multithreaded applications,” in ASPLOS IX Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, ACM press, New York, Nov 2000, pp. 117–128.