International Journal of Leading Research Publication (IJLRP)

E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

Architectural and Performance Considerations for Java Applications on z/OS

Chandra Mouli Yalamanchili

chandu85@gmail.com

Abstract

Java applications executed on IBM z/OS platforms benefit from the mainframe infrastructure's reliability, scalability, and security. However, attaining optimal performance requires more than conventional tuning techniques. [1][6]

This paper presents a research-oriented examination of optimization strategies and architectural trade-offs relevant to Java workloads on z/OS. Rather than prescribing specific configurations, a spectrum of performance approaches is surveyed—these range from low-level JVM parameter adjustments to high-level architectural and subsystem integration strategies.

This paper evaluates the execution contexts of both CICS-integrated and standalone Java applications, examining the roles of z Integrated Information Processors (zIIPs), garbage collection behaviors, thread models, file input/output practices, and DB2 connectivity models. Attention is also given to the interplay between resource efficiency and development agility, positioning performance choices as critical architectural decisions.

This paper aims to provide a structured discourse on Java performance strategies within the z/OS ecosystem through case studies, tool assessments, and conceptual models.

Keywords: Java on z/OS; JVM tuning; CICS Java; zIIP optimization; Mainframe performance; Liberty server; Batch processing

1. Introduction

In contemporary enterprise computing, where digital transformation increasingly depends on hybrid infrastructure models, the execution of Java on IBM z/OS systems is recognized as a strategic enabler of modernization. Java has been incorporated into z/OS environments to support RESTful API development, microservice orchestration, and batch processing frameworks. Nonetheless, stringent performance requirements—particularly within financial, healthcare, and high-availability systems—necessitate a nuanced approach to workload optimization. [1][7]

Rather than functioning as a prescriptive guide, this paper examines the landscape of performance considerations and optimization options available to Java applications on z/OS. Various configurations and trade-offs are explored and evaluated across multiple dimensions, including scalability, operational complexity, and cost-efficiency. Performance characteristics are reviewed within both CICS Liberty environments, where Java EE applications interact with traditional transaction processing subsystems, and in standalone contexts involving BPXBATCH or Unix System Services-based batch jobs. [2][6]

The focus of this analysis is not confined to technical configurations but extends to broader architectural and subsystem-level implications. Optimization is a strategic consideration requiring careful alignment between software behavior and platform capabilities. In this context, a structured evaluation is provided to assist system architects and enterprise developers understand the parameters and mechanisms that govern Java performance on z/OS.

2. Overview of Java on z/OS

The deployment of Java applications on IBM z/OS has emerged as a strategic approach to modernizing mainframe-based systems, while preserving the continuity and integrity of legacy COBOL and Assembler workloads. The IBM SDK for Java on z/OS, a certified implementation of the Java platform, has been specifically optimized for z/OS and is designed to integrate with key subsystems, including CICS, DB2, and MQ. This section presents an overview of the supported environments, architectural synergies, and workload patterns typically associated with Java on z/OS. [6][7]

Java workloads are supported across multiple runtime environments, including:

- **CICS Liberty JVM Server** supports transactional Java EE applications within a tightly integrated, secure execution context. [8]
- **Batch Java environments**, such as BPXBATCH, JZOS, and WebSphere Compute Grid, are predominantly used for high-volume data processing. [7]
- **Unix System Services (USS)** is the foundational environment for shell-initiated and batch-executed Java applications, including interactive utilities and automated job executions.

2.1. Platform-Specific Strengths

Several characteristics inherent to z/OS contribute to performance advantages when running Java applications:

- **z Integrated Information Processors (zIIPs)** facilitate cost-effective execution of eligible Java workloads by offloading processing from general-purpose CPUs. [6]
- **Shared Class Cache** enables the reuse of loaded classes across multiple JVMs, thereby reducing startup latency and memory duplication.
- **SMF (System Management Facility) Integration**, providing granular visibility into JVM usage, performance metrics, and billing data. [7]
- **Access to System-Level Resources**, including DB2 databases, MQ queues, VSAM datasets, and JES queues through optimized APIs.
- **Support for hybrid deployment options**, including z/OS Connect, zCX, z/VM, and zLinux environments, which allow Java components to be containerized or run under Linux guests on IBM Z. [6]

These capabilities enable Java applications to operate as fully integrated components within the mainframe execution environment.

2.2. Integration with Subsystems

Integration with traditional subsystems is a core strength of Java on z/OS. Native support is available through:

- **CICS Liberty JVM Server** allows Java EE applications to run within the same region as traditional COBOL programs while preserving transaction security and routing semantics. [4][8]
- **DB2 JDBC drivers**, which have been optimized for z/OS, offer improved performance and reduced CPU overhead during database interactions. [2]
- **MQ Java APIs** provide secure, asynchronous messaging capabilities between Java modules and existing z/OS applications. [6]

These integrations ensure that Java applications can participate in transactional flows and messaging infrastructures without disrupting operational patterns. Features like JCICS APIs allow Java programs to perform operations like VSAM access and inter-program communication directly within CICS regions. [8]

2.3. Workload Characteristics

Java on z/OS is employed for a range of workload scenarios, including:

- Exposing legacy business logic via RESTful interfaces.
- High-volume batch processing involving DB2, VSAM, or flat files.
- Workflow orchestration and composite service construction involving multiple back-end systems.
- Running modular OSGi-based applications or full Java EE workloads under Liberty for transactional or service-oriented use cases. [8]

In most scenarios, Java is not introduced as a replacement for traditional programs, but rather as a complementary layer that enhances modularity, portability, and cloud readiness. This co-existence strategy is particularly evident in phased modernization efforts, where Java and COBOL components communicate through the Language Environment (LE) and JNI layers. [7]

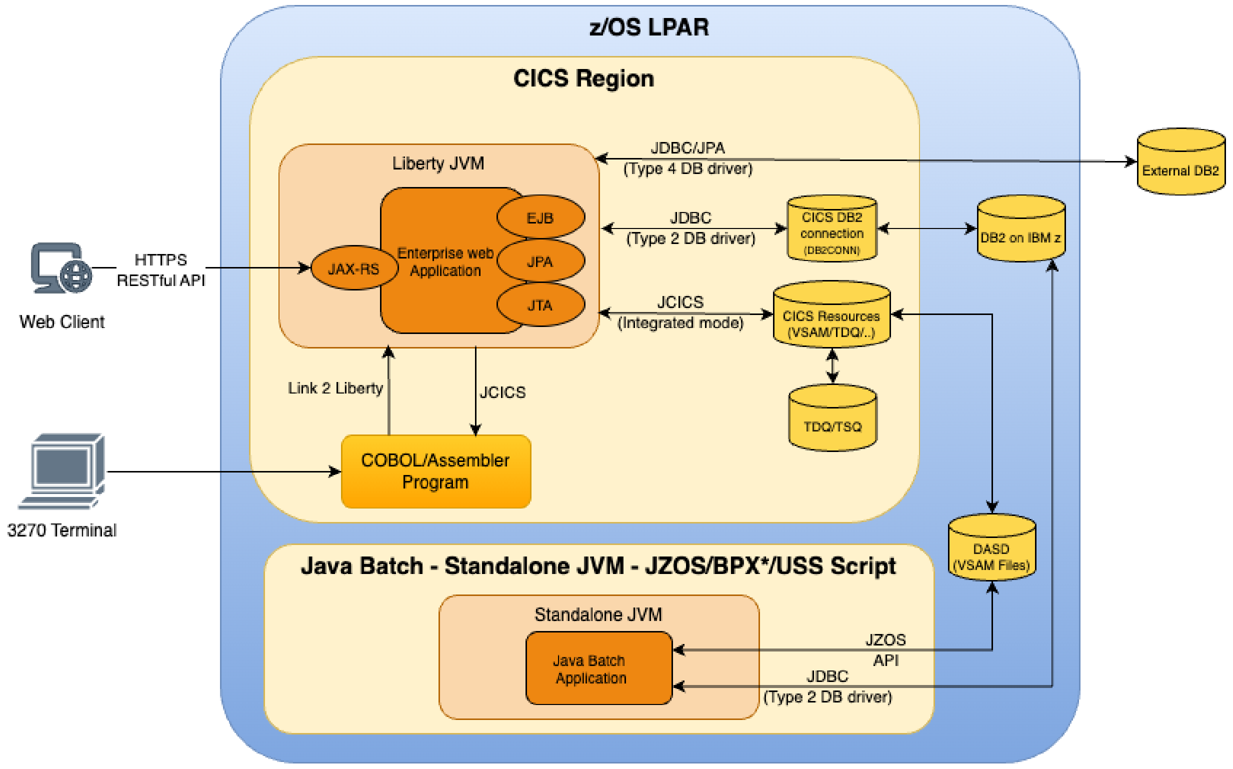

Figure 1: Architecture diagram illustrating different Java runtimes on z/OS. [2][3][6][8]

By capitalizing on existing hardware infrastructure and subsystem integration points, Java on z/OS provides a flexible and extensible foundation for application modernization. As will be examined in subsequent sections, the characteristics and constraints of each runtime environment directly influence performance optimization strategies. [1][3][6]

3. JVM Tuning for z/OS Performance

The performance of Java applications on z/OS is significantly influenced by the configuration of the Java Virtual Machine (JVM). This section outlines tuning concepts applicable across both CICS and non-CICS environments while avoiding repetition of details discussed in later sections. JVM tuning considerations span heap sizing, garbage collection (GC) policies, class cache sharing, thread pool management, and interaction with system-level resource controls such as zIIPs and WLM. [1][4]

3.1. Heap and Memory Management

JVM heap size and structure influence application stability and throughput. Key parameters typically include:

```
-Xms512m -Xmx2048m
-Xmn512m
-Xgcpolicy:gencon
-Xverbosegclog:gc.log
```

The gencon policy is recommended for transactional and batch Java workloads due to its balanced approach to GC pause times and throughput. [1][7] These settings broadly apply whether the JVM runs under Liberty in CICS or standalone under JZOS or USS.

3.2. zIIP Offloading

The JVM on z/OS is designed to offload eligible workloads to z Integrated Information Processors (zIIPs), improving cost efficiency. Enabling offloading requires:

```
-XX:+UseZEnterpriseCapabilities
```

This option is transparent to developers but crucial in high-volume environments. zIIP effectiveness has been demonstrated across Liberty, JZOS, and batch execution environments. [6][7]

3.3. Class Sharing and Startup

The shared class cache allows JVMs to reuse common bytecode, reducing class loading time:

```
-Xshareclasses:name=SharedCache,nonFatal
```

This feature enhances startup performance for both CICS Liberty and standalone batch jobs. Cold-start JVMs, such as those invoked repeatedly via BPXBATCH or JZOS, benefit significantly from this optimization. [4][6]

3.4. Thread Pool and Concurrency Management

Thread pools must be tailored to the workload model. While Liberty uses XML-based thread configuration, standalone Java uses programmatic constructs:

```
ExecutorService exec = Executors.newFixedThreadPool(20);
```

Thread pool sizing should reflect processor availability, WLM policy constraints, and transaction concurrency requirements. [1] Improper tuning can cause contention or underutilization.

3.5. Native Heap and Virtual Storage Limits

The system addresses space constraints that govern Native heap usage on z/OS. Monitoring parameters include:

```
-Xdump:heap:events=user

-Xdump:system:events=user
```

In both CICS and non-CICS use cases, SMF records (e.g., Types 30 and 89) assist in analyzing JVM memory behavior. JVM tuning should align with workload classification rules defined in WLM. [3][7]

3.6. Observability and Dynamic Adjustments

JVM configurations must be adaptable to workload patterns. Tools like IBM Health Center, JMX monitoring, and verbose GC logs provide runtime feedback essential for tuning decisions. [3][5] These tools apply across Liberty and standalone JVMs and are integral to iterative performance refinement.

A foundational understanding of JVM behavior is essential for optimizing Java applications on z/OS. While deeper integration specifics are handled in Sections 4 and 5, the strategies summarized here establish a consistent and scalable tuning baseline across all Java environments on the platform. [1][4][7]

4. Performance Optimization for CICS-Embedded Java

CICS Transaction Server supports the deployment of Java applications using the Liberty JVM server, a lightweight Java EE container optimized for transactional environments. Java execution in CICS must comply with stringent performance requirements associated with high-throughput online transaction processing (OLTP). This section analyzes the tuning strategies and operational considerations necessary for effective performance management of Java workloads embedded within CICS regions [1][4].

4.1. Liberty JVM Server Configuration

Liberty JVM servers within CICS regions are configured using a combination of jvmprofile.properties and server.xml files. Critical JVM options typically include:

```
-Xms1024m
-Xmx2048m
-Xgcpolicy:gencon
-Xshareclasses:name=CICSAppCache,nonFatal
```

These settings have been shown to improve cold start time, reduce memory churn, and minimize garbage collection (GC) latency [1][4].

Thread pool and HTTP dispatcher tuning are handled in Liberty's server.xml:

```
<executor name="defaultExecutor" coreThreads="30" maxThreads="100" />
<httpDispatcher maxConcurrentRequests="500"/>
```

These values must be calibrated with transaction volume and the CICS region's thread limits (typically up to 256 concurrent threads per Liberty JVM server) [4].

4.2. Application Design for Performance

Optimizing Java applications within CICS requires careful attention to architectural patterns:

- **Asynchronous Design**: Non-blocking APIs (e.g., @Asynchronous, CompletableFuture) are recommended to prevent thread exhaustion. [6]

- **Efficient Resource Access**: Connection pools should be managed using JCA adapters for DB2, MQ, and external services.
- **Scope Minimization**: CDI beans should default to @RequestScoped or @Dependent to avoid excessive memory retention.
- **Minimized Object Footprint**: Payloads exchanged via JCICS containers or EXEC CICS LINK should avoid deeply nested structures to reduce serialization overhead. [8]

4.3. Interoperability with COBOL/Assembler

Java components frequently interact with traditional programs in CICS:

- Calls from COBOL to Java are facilitated using EXEC CICS LINK, with communication handled via channel containers. [2]
- Java-to-COBOL invocations are performed using the JCICS Program class, with byte-oriented data exchange advised for high-volume tasks.
- Reusing established containers and files (e.g., VSAM, TSQs) should follow the JCICS best practices to avoid unnecessary open/close cycles. [8]

4.4. Security and Identity Propagation

Identity propagation ensures consistent access control and auditing:

- Liberty server configurations should include `<authData alias="zosUser"/>` entries for the mapped identity. [4]
- Liberty can enforce Java EE security roles mapped to RACF groups using server.xml role bindings.

4.5. Thread and Memory Constraints

CICS enforces specific threading and heap usage limitations:

- Each Liberty JVM server supports up to 256 threads; total region-level threads typically do not exceed 2000 across all servers and tasks. [8]
- JVM heap sizes must be tuned to avoid excessive GC while respecting region boundaries (e.g., -Xms512m -Xmx2048m). [1]
- Diagnostic tuning via -Xdump:system and -Xverbosegclog aids in correlating memory use with transaction throughput. [3]

4.6. Monitoring and Operational Best Practices

- Liberty metrics can be monitored via JMX-based interfaces and SMF reporting mechanisms.[3][4]

- SMF type 88 and 120 records provide insight into JVM usage at both the task and region levels. [3]
- Integration with tools like CICS Explorer and Health Center facilitates runtime visibility and proactive tuning. [6]

4.7. Additional Tuning Recommendations

- Place the JDBC drivers and common libraries in shared classloaders to reduce memory overhead. [1]
- Avoid excessive classpath scanning using explicit configuration files (web.xml, beans.xml).
- Maintain JVM server warmth by minimizing startup-shutdown cycles; persistent servers improve response time consistency. [4]

A well-tuned Liberty JVM server within CICS provides the necessary performance profile for modernizing transaction-heavy applications while leveraging the reliability and security of the mainframe. When thread allocation, heap sizing, and identity propagation are configured with workload patterns, Java components can operate alongside COBOL and Assembler programs without introducing contention or instability. [6]

5. Performance Optimization for Standalone and Batch Java

In addition to CICS-based applications, Java is frequently utilized in standalone modes on z/OS to perform background computation, file manipulation, and high-volume batch processing. These workloads are typically executed under environments such as Unix System Services (USS), BPXBATCH, or JZOS, and often involve integration with traditional JCL and MVS datasets. This section presents performance optimization strategies specifically applicable to these environments, drawing on real-world implementations and IBM guidance. [1][2]

5.1. Runtime Environments

The following deployment models are commonly used for standalone Java execution on z/OS:

- **BPXBATCH**: Executes Java programs in a UNIX System Services shell as a child subtask, offering a lightweight option without JCL dataset integration. [7]
- **BPXATSL**: Similar to BPXBATCH but runs in the main task, enabling access to JCL DD statements and job control resources.
- **JZOS Batch Launcher**: A preferred method for Java batch jobs requiring full JCL integration, including DD name access and EBCDIC-aware dataset handling. [2]
- **USS Shell Execution**: Direct command-line invocation, often used for automation or testing.

Each model presents unique dataset access, logging, and integration complexity trade-offs. A comparative selection matrix based on resource usage, JCL access, and monitoring support is available in IBM documentation. [7]

5.2. File I/O and Dataset Handling

File handling efficiency is critical in Java batch workloads:

- Use buffered I/O to reduce overhead.
- Prefer NIO-based FileChannel APIs for large file throughput for USS files.
- For MVS datasets, leverage JZOS APIs such as ZFile to interact with sequential or VSAM files in record mode. [7]

```
ZFile zfile = new ZFile("//DD:INPUT", "rb,type=record");
BufferedReader reader = new BufferedReader(new InputStreamReader(zfile.getInputStream(),
"Cp1047"));
```

Encoding differences (ASCII/Unicode vs. EBCDIC) must be addressed using appropriate code pages (e.g., Cp1047) to avoid data corruption or logic errors in I/O handling. [2]

5.3. Multi-Threaded Batch Jobs

To achieve higher throughput in compute-intensive tasks:

- Employ fixed or bounded thread pools to process record segments in parallel.
- Use data partitioning to divide the workload based on file offsets, keys, or logical blocks. [1]

```
ExecutorService exec = Executors.newFixedThreadPool(10);
for (Chunk c : chunks) {
   exec.submit(() -> processChunk(c));
}
```

Thread count should be aligned with the number of available processors and WLM dispatching priorities to avoid resource contention. Thread-local JDBC connections or pooled resource managers are recommended for data access tasks. [2]

5.4. Database Optimization

When accessing DB2 from batch Java:

- Reuse PreparedStatement instances and avoid unnecessary connection opening.
- Set fetchSize and maxRows to control memory consumption.
- Validate DB2 indexing and query plans using EXPLAIN support. [2]

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM TXN WHERE TS > ?");

ps.setFetchSize(500);
```

JZOS-based JDBC applications may benefit from RACF pass-through authentication and DSNHDECP configuration for improved efficiency. [7]

5.5. Resource Management and Exit Handling

Batch applications must ensure graceful shutdown of system resources:

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> cleanup()));
```

This hook can release file handles, flush logs, or close JDBC sessions. Failure to implement proper exit handling may lead to dataset contention or orphaned output files in the JES2 spool. [3]

5.6. Integration with COBOL and JCL

Java batch programs can be integrated into traditional JCL-based flows:

- Use the JZOS MVSJobStep interface to interact with SYSIN, SYSPRINT, and DD-defined datasets. [2]
- DD names are passed into the JVM via environment variables or STDENV.

```
//JAVAEXEC EXEC PGM=JVMLDM86,PARM='com.batch.Main'
//STEPLIB  DD DISP=SHR,DSN=IBM.JZOS.LOADLIB
//INPUT    DD DISP=SHR,DSN=MY.INPUT.DATASET
//STDENV   DD *
JAVA_HOME=/usr/lpp/java/J8.0
CLASSPATH=/u/apps/batch.jar
/*
```

Batch applications may also interoperate with COBOL modules through JNI or by coordinating output datasets processed in downstream steps. [6]

5.7. Monitoring and Operational Best Practices

- **SMF Type 30**: Collects job-level CPU, memory, and I/O metrics. [3]
- **SMF Type 89**: Tracks JVM usage by service class for workload classification.
- **System logs**: USS stderr and stdout should be routed to JES SYSOUT or USS log files.
- **IBM Health Center** and -Xverbosegclog can be used to monitor GC and thread behavior. [3]

5.8. Summary of Optimization Recommendations

- Align JVM heap and GC policy with expected batch sizes. [1]
- Use JZOS APIs for MVS dataset compatibility and character set conversion. [2]
- Tune thread pool sizes based on WLM and processor availability.

- Profile SMF records to validate performance improvements. [3]

Batch Java programs, when tuned properly, can match or exceed the efficiency of legacy COBOL batch jobs. Their modularity, ease of integration, and use of modern libraries are critical components in hybrid modernization strategies for z/OS environments. [6]

6. Monitoring and Diagnostic Tools

Performance tuning is only effective when supported by continuous monitoring and diagnostics. On z/OS, various system and JVM-level tools exist to observe and analyze Java application behavior. These tools aid in identifying memory leaks, GC inefficiencies, threading bottlenecks, and I/O contention, forming the foundation of a sustainable performance management strategy. [1]

6.1. IBM Health Center and Diagnostic Tools

IBM Health Center, part of the IBM Monitoring and Diagnostic Tools package, enables detailed insight into JVM activity:

- Garbage Collection behavior
- Object allocation rates
- Live thread states
- Lock contention

Monitoring data can be collected in real time or after execution using:

```
-Xhealthcenter:level=all
```

Analysis is typically performed through an Eclipse-based or headless UI to support both development and production environments. [1][3]

6.2. SMF Records and System Insights

The System Management Facility (SMF) provides vital statistics on workload behavior:

- **SMF Type 30**: Captures job-level resource usage, including CPU and memory.
- **SMF Type 88/120**: Reports on Liberty JVM servers within CICS.
- **SMF Type 89**: Classifies JVM usage by service class and shared cache metrics.

These records are analyzed using RMF, SAS, or custom SMF parsers for trend evaluation and exception detection. [3]

6.3. Java JMX and Liberty-Specific Monitoring

For applications running under Liberty, especially in z/OS environments such as CICS Liberty JVM servers, JMX remains the primary and most reliable mechanism for exposing runtime metrics:

- JMX ports can be configured and accessed using tools like VisualVM, JConsole, or custom JMX consumers.
- JMX-based monitoring allows memory usage, thread activity, and garbage collection behavior to be observed.

While MicroProfile features like mpMetrics are supported in general Liberty distributions, their availability and applicability in CICS Liberty configurations may be limited or unsupported in certain contexts. Therefore, operational monitoring should emphasize supported mechanisms like JMX and SMF-based insights for Liberty on z/OS. [3][4]

6.4. z/OS System-Level Monitoring

In addition to JVM-specific tools, broader system observability can be achieved through:

- **RMF (Resource Measurement Facility)**: For CPU dispatching, storage paging, and DASD utilization.
- **OMEGAMON for JVM/CICS/z/OS**: Offers real-time dashboards and alerting mechanisms.
- **CICS Explorer**: A GUI-based interface to review Liberty JVM server health, monitor SMF metrics, and visualize thread pool activity. [4]

6.5. Logging and Application Telemetry

Structured logging facilitates correlation across distributed components and z/OS services:

```
logger.info("Processed {} records in {} ms", count, elapsed);
```

Enterprise observability tools such as Splunk or the Elastic Stack can collect logs from Liberty servers and USS environments. These tools support pattern detection, anomaly correlation, and compliance auditing. [2][6]

Proactive monitoring and consistent logging support faster diagnosis, better resource planning, and early anomaly detection. Integrating these tools into the application lifecycle is essential for maintaining optimal performance on z/OS. [1][3]

7. Case Studies and Lessons Learned

The following examples illustrate two representative Java use cases on z/OS—one based on a CICS-integrated transactional service, and another based on a batch-oriented workload. These conceptual use cases demonstrate how tuning and architectural decisions can address common performance challenges in z/OS Java environments.

7.1. Use Case: CICS Liberty Application Optimization

A Java REST API is deployed within a CICS Liberty JVM server to modernize access to a legacy COBOL transaction in this scenario. For this scenario, the challenges considered are high response time and insufficient for high-throughput transactional workloads.

Optimization Steps:

- Enable -Xshareclasses to improve JVM startup and reduce class loading overhead.
- Tune thread pool configuration in server.xml to align with peak transactions per second (TPS).
- Replace blocking I/O operations with non-blocking constructs using CompletableFuture.
- Implement database connection pooling via the CICS JCA DB2 adapter. [4]

Outcome:

- Reduction in overall response time.
- Liberty servers scaled horizontally across multiple JVM instances.
- zIIP offloading led to a reduction in general CPU usage. [4]

7.2. Use Case: Batch Java Job on BPXBATCH

This example considers a batch job that processes over 10 million daily call data records using Java under BPXBATCH, replacing a legacy COBOL implementation.

Challenges:

- VSAM file access incurred high latency.
- JVM heap pressure caused frequent garbage collection.
- Aggregation logic introduced memory bottlenecks.

Solutions:

- Adopt parallel processing using a ForkJoinPool to partition the workload.
- Switch from unbuffered to buffered reads using Java NIO channels.
- Tune JVM heap size (-Xmx) to match region capacity and prevent paging. [2][7]

Outcome:

- Reduction in processing time.
- The workload is offloaded to zIIP processors, lowering CPU billing.
- Enhanced maintainability and reduced onboarding time for development teams.

7.3. Lessons Learned

- **Thread pool tuning is critical**: Both under- and over-provisioned thread configurations degrade performance in transactional and batch workloads.

- **Monitoring is continuous**: Garbage collection, memory utilization, and thread usage fluctuate with data volume and input patterns, requiring iterative tuning.
- **Subsystem configuration matters**: DB2 fetch sizes, Liberty startup settings, and JCL integration directly impact throughput and reliability. [1][3]

These conceptual use cases highlight how Java performance on z/OS can be significantly improved through strategic tuning and architectural decisions. Though hypothetical, they reflect patterns frequently encountered in modernization projects and demonstrate that Java can meet or surpass the performance traditionally delivered by COBOL-based systems when optimized effectively. [1][2][4]

Conclusion

A research-based examination of the optimization considerations applicable to Java applications on IBM z/OS has been conducted throughout this paper. By evaluating tuning techniques, subsystem interactions, and configuration strategies, Java performance was framed not as a static checklist, but as a dynamic, context-sensitive architectural concern. Within CICS regions, Java introduces both latency challenges and architectural flexibility, particularly in deploying transactional APIs. In standalone batch environments, performance optimization involved memory tuning, thread control, and I/O throughput alignment. [2][7]

Harmonizing JVM capabilities with mainframe architectural constructs remains central to achieving reliable and cost-effective execution of Java workloads. As system complexity and hybrid integration patterns continue to evolve, the scope of optimization will expand further. Therefore, future research should explore autonomic workload tuning, AI-based performance analytics, and formal modeling frameworks tailored to the characteristics of Java on z/OS. The role of Java in mainframe environments is thus positioned not only as a tool for modernization but as a platform for sustained research and innovation in enterprise software performance. [1][6]

References

[1] IBM Corporation, "IBM SDK, Java Technology Edition", IBM Documentation. [Online]. Available: https://www.ibm.com/docs/en/sdk-java-technology/8

[2] IBM Corporation, "Batch Modernization on z/OS ", IBM Redbooks, SG24-7779-01, July 2012. [Online]. Available: https://www.redbooks.ibm.com/abstracts/sg247779.html

[3] IBM Corporation, " Tuning the IBM virtual machine for Java",IBM Documentation. [Online]. Available: https://www.ibm.com/docs/en/was-zos/9.0.5?topic=jvm-tuning-virtual-machine-java

[4] IBM Corporation, "Liberty in IBM CICS: Deploying and Managing Java EE Applications", IBM Redbooks, SG24-8418-00, January 2018. [Online]. Available: https://www.redbooks.ibm.com/abstracts/sg248418.html

[5] IBM Corporation, " IBM z15 Technical Introduction", IBM Redbooks SG24-8852-00. August 2020. [Online]. Available: https://www.redbooks.ibm.com/abstracts/sg248852.html

[6] C. Yalamanchili, “Java on IBM z/OS: Deployment Models and Technical Considerations,” IJSAT, vol. 14, no. 2, Apr. 2023.

[7] C. Yalamanchili, “Evolving Mainframe Batch: Java Workload Strategies on z/OS,” IJLRP, vol. 4, no. 8, Aug. 2023.

[8] C. Yalamanchili, “Technical Insights into Running Java Applications in CICS,” IJIRMPS, vol. 11, no. 5, Sep.–Oct. 2023.