H-0250 (H0602-017) February 13, 2006
Computer Science

IBM Research Report

A High-Performance Domain Specific Parallel and Distributed Massive Collection System

Uri Shani, Aviad Sela, Inna Skarbovsky
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A High-Performance Domain Specific Parallel and Distributed Massive Collection System

Uri Shani
IBM Haifa Research Lab
Haifa University Mount Carmel
31905 Haifa, Israel
+972 4 8296282
shani@il.ibm.com

Aviad Sela
IBM Haifa Research Lab
Haifa University Mount Carmel
31905 Haifa, Israel
+972 4 8296456
sela@il.ibm.com

Inna Skarbovsky
IBM Haifa Research Lab
Haifa University Mount Carmel
31905 Haifa, Israel
+972 4 8296569
inna@il.ibm.com

ABSTRACT

High performance and ease of use are the two main goals of the Massive Collection System (MCS). On the outset, MCS is a classical process that consumes massive amount of input, processes it according to business specifications, and produces a comparable amount of output. To do that, MCS has a massive parallel architecture whose core processing task executes the business rules on a continuous flux of input records organized in files. Each processing task executes a processing "plan" which is a high level domain specific language (DSL) designed for domain experts rather than professional programmers.

The MCS design for performance is composed of two factors: one is the massively parallel execution framework; the second is the effective compilation and execution of the domain specific MCS plans. The execution framework is built on top of IBM J2EE implementation Websphere Application Server (WAS). The entire MCS is a WAS application, written in Java, which obtained its performance goals as well as ease of use.

The performance challenges of MCS were stated in terms of hundreds of millions of records a day. We selected Java and WAS for implementation due to their development advantages, allowing us to obtain proofs for the MCS performance goals rather early – within several months, which were shown to scale up almost linearly on the input size.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications – *Specialized application languages, Very high-level languages;* D.3.3 [**Programming Languages**]: Processors – *Run-time environments;* D.2.6 [**Software Engineering**]: Programming Environments – *Integrated environments;*

General Terms

Management, Measurement, Performance, Design, Reliability, Experimentation, Languages.

Keywords

Scalable Middleware, Massive Records Collection, Massive event processing. Revenue assurance, Distributed Records Collection

1. INTRODUCTION

Many business operations generate large amount of event records describing the business progress, such as banking and other financial institutes. MCS was designed to handle a high volume of event records in the Telco industry where these are Call Detail Records (CDRs) describing individual telephone call events. The MCS end to end process in the Telco context is as follows:

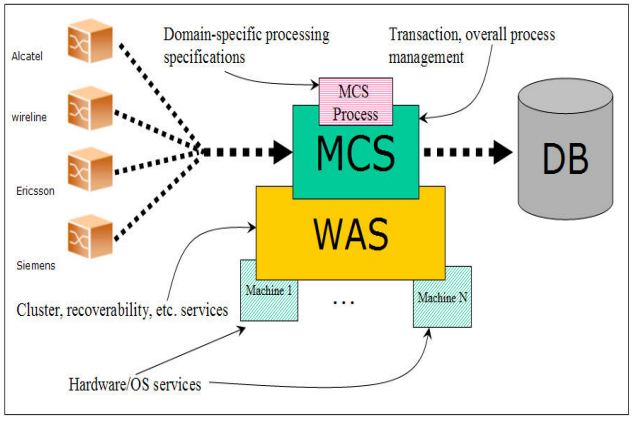

Figure 1-1: MCS layered architecture

The records originate in the operational network in switches, and are pushed upstream into central processing where eventually they drive billing systems which when materialize, generate the service provider revenues. MCS was specifically required to handle a revenue assurance function, by which it processes CDRs in parallel to the billing system, repeating its function in order to find mismatches in the data which may lead to the discovery of revenue leakages.

To facilitate effective system, the rules by which CDRs are processed should be easily written by domain experts rather than programmers. That requirement led to the design of a domain specific language (DSL) [1] in which the business rule specifications for each type of CRD were coded.

We will first describe the MCS architecture, which is composed of two major layers; The MCS framework is the execution environment layer in which MCS plans are processed rapidly and highly parallel. The architecture will describe how the plans' execution is orchestrated on top of the IBM Websphere Application Server (WAS) [2],[3],[4], which is the IBM J2EE [5] implementation.

The MCS performance is measured along two dimensions. On the local scale, each group of records is processed by a certain processing plan instance and must reach a certain high level of performance to meet end to end demands. The collection of such executions is processed in a highly parallel framework and presents the overall end to end MCS performance which processes a mix of input record streams.

We will also discuss future work in which the typical performance characteristics of an MCS-like environment are identified and contribute to an effective model by which MCS performance can be predicted and measured.

2. MCS Architecture

MCS architecture is designed to efficiently address the following aspects:

1. A sequence of execution steps among which are the following main processing requirements :
    a. Process a group of input records of a certain type according to a certain processing plan
    b. Storing the output records either in a database or file system.
2. Orchestrating the massive execution of these processes in a parallel and distributed J2EE application.

2.1 The MCS framework

2.1.1 MCS Entities

An MCS '*Entity*' is represented by a *Data Object* (*DO*) persistent by MCS database tables. MCS allows many entity instances of the same type. The DO is responsible for maintaining the state of its associated entity instance. Each entity instance has a defined life cycle managed by a state-machine object. The lifecycle is defined by a state machine diagram, which is further described in the following subsection.

MCS apportions the massive input flux into a manageable quantity which is called an '*Envelope*'. The envelope forms a unit of work (UOW) on which MCS ensures that an end-to-end global process either completes or fails all together. A MCS envelope consists of one or more '*Task*'s, where each task represents a sub set of the input records assigned to the owner envelope. Generally, MCS aggregates input records into one or more files, where each file (also termed "Fragment" file) is associated with its own task. The total number of records in all these files defines the UOW. Both envelopes and tasks are MCS entities.

MCS tasks can be processed anywhere in the system independent of other tasks. A MCS task is responsible for invoking the MCS language processor responsible for processing the records according to a defined plan written specifically to the input record's designated type. The task entity reads the input records from the assigned source (e.g., file, input stream etc..), processes the records according to the plan and ends up with a collection of output records associated with a target destination which is a property of the owner envelope.

MCS Destination is a logical entity which is associated with a real world resource, which in general will be a database table or a file. The task creates an '*OutputFile*' entity whose DO maintains the state of the concrete destination instance, throughout their lifecycle from generation by a task, to completion with a successful database uploading or safely stored in a file on a file system.

The MCS massive uploading to target table destination is carried out by the MCS "*Package*" entity. MCS manages one or more package entities each one is a singleton in the system responsible for uploading a specific MCS target destination. A package may upload data produced by different envelopes sharing the same MCS destination.

MCS also defines the '*Poller'* entity, a system wide singleton, responsible 'sniffing' the file system for incoming input records, issuing new job request for the MCS envelopes.

The MCS interaction is illustrated in the following relations diagram:

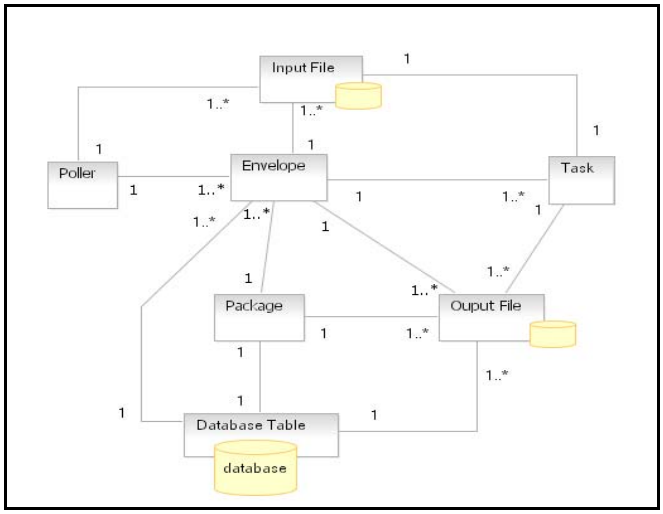

Figure 2-1: MCS Entities relations

The poller, envelopes, tasks and packages entities shown in the above figure are processing entities, meaning MCS executes a thread for each execution step in their lifecycle. The output file which is a product of the task process is subject for further management by the package and the envelope entities. The other entities, such as the '*InputFile*' and the '*Database Table*', do not have specific MCS entity representation, except as a real world

entities existing in some media (e.g. file system and database, respectively).

2.1.2 MCS Flow

MCS manages the life cycle of its entities using a state machine object created by MCS Business Process Flow & Control Engine (BPFC) according to a defined State Machine Diagram. The state machine object maintains the state of each entity instance within its associated DO instance. The MCS BPFC engine is responsible for providing transaction support for any state transition and any action taken against external and internal resources such as database tables or messaging engines queues.

The MCS flow is driven by messages, that MCS calls '*Trigger',* sent by entity instances within state machine actions. The triggers can be sent internally targeted to the same entity or to other entity types. Triggers are dispatched through queues defined in the messaging layer provided by WAS. The BFPC engine reads the triggers from the queue (via Message Driven Beans - MDBs) and deciphers it to identify the target entity DO, and retrieve it from the database. This DO, associated with the corresponding state-machine diagram uniquely recreates the entity's state at the end of its previous processing step – the state-machine instance object. Then the engine propagates both the retrieved DO and the incoming trigger to the deciphered state-machine object for the next actual processing step.

State-machine processing depends on the persistent DO state and the incoming trigger ID. If no match exists for the trigger ID with the current state according to the state-machine diagram, the trigger is discarded. Otherwise, a state-machine '*Transition Action*' is taken followed by a '*State Flow Processes Action*'. Basically, a transition action is a 'short term' (e.g. in order of milliseconds) consists of three successive actions: starting state *Exit* action, actual transition action and the *End* state action. This sequence of actions is transactional, persistent in the database on success or rolled back on failure. On the other hand, the state flow action is a 'long term' (e.g. in order of seconds/minutes) action, that programmatically addresses failures. Meaning, on failure the state flow action issues a required trigger, to drive the entity to a handling state.

MCS business processing is carried out by the flow action implementations. Specifically, the input record processing is carried out by a state flow action in the task's state machine diagram, and database uploading is a state flow action of the package state machine diagram. As described above, each actual processing step is carried out within a WAS-assigned execution thread.

For each of the four processing entity types MCS defines a state machines diagram: poller, envelope, task and package. Each entity instance operates as a J2EE MDB, listening for incoming triggers on the WAS messaging queues. When deployed as a cluster of servers on one or more machines, the message engines on each of the cluster nodes ensures load balancing via the Workload Management (WLM) feature of WAS. The scope of this paper is too short to provide the needed details which can be found in [2], [3], [4].

2.2 The MCS task processing

MCS language is modeled in UML [6], and converted to an executable module using the Eclipse Modeling Framework (EMF) [7],[8]. With that tool, MCS obtains a highly effective integrated development environment (IDE) as an MCS language plugin to Eclipse [9], [10]. Figure 2-2 artistically shows two panels of this IDE, where the lower is an imperative processing plan in edit through the EMF generated tree visual editor. In that pane, a "Table" element is in focus, and whose properties are seen on the upper pane, depicting a certain table and columns in a relational database.

The result of this IDE is an MCS specification plan file in XML[11] format. The file is in fact an instance of a model of the language, rather than a commonly ASCII representation of a programming language, and for that we use the XMI[12] standard language for XML model interchange.

.

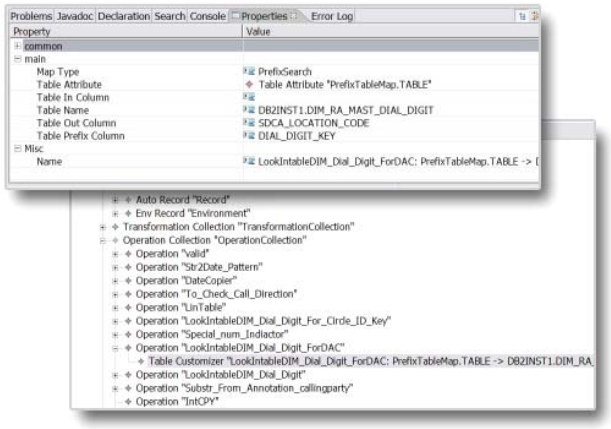

Figure 2-2: The MCS integrated development environment over the Eclipse platform, using the Eclipse Modeling Framework.

The language domain is the processing of records consisting of fields, having certain types, coming from an input file. To convert input records to output, the MCS processing plan performs a set of transformations, each of which has an underlying "Base Operation". Base operations are MCS language elements which interface the task execution framework to Java classes which are written according to a certain MCS standard interface. While we do not have the space to go into the MCS language constructs in detail, it should be clear that at execution time, base operations are invoked in sequence to compute new values from input values and produce them in the output fields of output records.

The execution model of a task is therefore the following pseudo code:

```
while (! EOF(input))
{
  read (input, inputRecord);
  clear ( outputRecord)
  for each trans in (transformations)
    trans.invoke(inputRecord, outputRecord);
  if (outputRecord.isValid())
    write(output, outputRecord);
}
```

This is an implicit control flow in MCS, which mimics that of the AWK language [13]. This is also an oversimplification of the MCS plan execution flow, but shows that input processing depends on the number of fields in input and output records, and number of transformations of invocations of base operation. Each base operation may have different processing complexities, the simplest of which are doing simple type conversions and content copying, whereas the more complicated have to do with table lookups, such as are done via the Table language element shown in Figure 2-1. Yet some other base operations can be very "heavy" as there is no limit on the complexity of each, except for the practical guidelines documentation in which such cases are not recommended.

3. MCS Performance steps

In the MCS design phases the requirements to support hundreds of millions of CDRs a day lead to the above architecture promoting performance issues to the reliable working of a highly parallel system. Yet, with individual task not performing a minimal load in reasonable time, the end to end process will not work on a reasonable hardware. A configuration of few machines, and some dozen or so processors has been considered, but not of dozens of machines with hundreds of processors. Converting the system planned daily intake of 500 million records, is equivalent to about 6,000 records a second (500,000,000/24/3600 = ~5,800). Therefore, processing in excess of 1,000 records a second in average was required.

3.1 The MCS task performance

3.1.1 The MCS task as a Java application

At the initial project phases, the use of Java has been preferred for 2 reasons. One is the language natural advantages as robust, reliable and easy to develop advanced tool. The second is our desire to enjoy the scalability, reliability and efficiency of the IBM WAS platform. With these environments it would be quicker to reach MCS goals. Therefore the initial test has been a Java program which does all steps in the above pseudo code: reading input, parsing and converting to internal representations, performing all transformations, and table lookups, and eventually write it all to an ASCII file, or alternatively uploading to a database table using JDBC [14] which is a standard Java API for doing SQL commands to a database management system – in our case: DB2. A second level testing wrapped this program as a WAS Message Driven Enterprise Java Bean (MDB) invoking it to compute same data within this framework to measure the effect it will have on reducing MCS performance.

Machines have been a desktop PC on which the test ran, and a database DB2 engine on a 4-way PC, reading 140 MB from 7 input files, each consisting of 20,000 records of 1KB each.

The results of this test have been very encouraging as in the following tables for single thread execution, which means a serial execution, thus the results are the average per file.

Legend: FR = file read time; Pr = Processing, En = Enrichment, Sum = Summaries – these are three types of business rules, so for this test we will look at sum of all of them. File column is for file output and DB is for database uploading. Time is in milliseconds, except for **totals** which are in seconds.

DB 4 CPU						
DB						
FR	Pr	En	Sum	DB	*File Total*	Total for all Files
38.714	677.57	1298	262	7726.9	*0:00:11*	0:01:15

Files- Local File system						
FR	Pr	En	Sum	File	*File Total*	Total for all files
38.71	677.57	1298	261.7	2818	*00:00:06*	0:00:43

Thus, the 20,000 records were processed in a total of 11 seconds, or 1,800 records per second, including database upload, and even better: 6 seconds for output written to files, or 3,300 records a second. On a RS/6000 (IBM P Series) server, running AIX 5.1, the following results were obtained:

Files- Local File System						
FR	Pr	En	Sum	File L	*File Total*	Total for all files
15	372	703	165	500.14	*00:00:02*	*00:00:13*

Here, we even have 10,000 records a second.

With multiple threads, results improve as follows for windows, ranging from single thread on 1st line, to 7 threads on 7th line:

Files- Local File system						
FR	Pr	En	Sum	File	*File Total*	Total for all files
38.71	677.57	1298	261.7	2818	*00:00:06*	0:00:43
32.86	683.14	1317	257.9	2376	*00:00:07*	0:00:39
38.57	1303.4	2065	362.1	2877	*00:00:08*	0:00:35
22.86	2010.3	2453	684.7	6026	*00:00:13*	0:00:28
51.57	2650.9	2733	511.1	6390	*00:00:12*	0:00:27
88.71	3900.4	4015	453.6	9020	*00:00:18*	0:00:28
1167	5068.6	5802	1326	14370	*00:00:30*	0:00:42

On AIX platform results are:

Files- Local File System						
FR	Pr	En	Sum	File L	*File Total*	Total for all files
15	372	703	165	500.14	*00:00:02*	*00:00:13*
15	468	837	191	566.71	*00:00:02*	*00:00:10*
17	656	1174	224	758.14	*00:00:03*	*00:00:09*
18	570	1068	240	973	*00:00:04*	*00:00:09*
19	753	1808	266	1050.4	*00:00:05*	*00:00:08*
25	919	2443	304	1197.7	*00:00:06*	*00:00:09*
27	1157	3008	331	1544.7	*00:00:07*	*00:00:08*

The best overall results on Windows are for 5-6 threads, as well as for AIX, with total performance of 140,000 records in 27 seconds on Windows, and 9 seconds on AIX, or 5,100 records per second, and 15,500 records per second respectively. These number surpass the original requirements by far, and leave sufficient margins for the expected more complicated real-world cases, in the yet-to be developed high-level language interpretation.

3.1.2 The MCS task using MCS language

The MCS IDE includes also tools to execute and measure performance of MCS plans. The execution environment provides access to all resources which would also be needed when executing on a server. In fact, it is possible to run an MCS plan within the IDE, accessing the exact same resources as on the

server. Processing time of plans excluding the compilation time will reflect very genuinely the processing time expected on the server.

On IDE we can measure several factors of the execution:

1. Execution time, separating read time from processing time, from output time.
2. Profile of the relative time consumption of each of the base operations, and their frequency of use
3. Data allocation and release on the Java heap as reflected in the Java garbage-collection (GC) reporting during execution.

3.1.3 Plan execution performance statistics

The following table summarizes the execution of a plan providing some performance insight as per #1 above:

Num records	1,081	Run #	Read	Processac	Harvest	Total	KR/sec
File size	490KB	1	n/a	n/a	n/a		
#Fields	119	2	301	340	300	941	1.15
#transformations	200	3	0	380	250	630	1.72
		4	0	301	230	531	2.04
		5	0	471	180	651	1.66
		6	0	390	150	540	2.00
		7	0	451	130	581	1.86
		8	0	331	240	571	1.89
		9	0	461	231	692	1.56
		10	0	461	161	622	1.74
Average (ms):			33.44	398.44	208.00	640	1.74
Per record (micro S):			30.94	368.59	192.41		
Per Transf (nano S):				1,842.94			
Light distance (Meter):				614.31			

As follows: The input file consisted of 1,081 records, executed 10 times, of which the first one included the plan compilation. File size is 490KB due to record size of about 0.5KB. Each record defines 119 fields, and the plan contains about 200 transformations. The average read/processing/output time per execution is (in milliseconds): 33.44/398.44/208, total of 640 ms/record, which is same as 1.74 K records/second. Further analysis per record is 368.59 µ-seconds, and further per transformation is 1.842 µ-seconds, or 1,842 η-seconds per transformation.

3.1.4 MCS Base operation profiling

The analysis above is the average per transformation which consists of data-field access, and performing some algorithm. Base operations may access from one to a dozen of data fields, so a good model for base operation performance is a bit more intricate, however the following results of profiling the base operations for task performance item #2 above shed more light on these fundamental elements of the MCS performance:

Grand Total	Grand Total	Base Operations	micros
-	0	LookupTableMap.	-
20.00	25000	BooleanFieldCopier.	0.80
150.00	25000	CalcEventTimeslot.	6.00
3,133.00	25000	MakeUniqueCDRKey.	125.32
10.00	25000	MinusDurationToDate.	0.40
20.00	50000	CompareBigger.	0.40
80.00	50000	NumberFormat.	1.60
50.00	50000	SubString.	1.00
501.00	74994	PrefixTableMap.	6.68
31.00	75000	AddTwoIntFields.	0.41
2,522.00	75000	StringToDate.	33.63
70.00	100000	DateCopier.	0.70
1,431.00	159911	LookupTableSet.	8.95
162.00	224990	OperatorEqual.	0.72
41.00	299976	TrueOperator.	0.14
220.00	415077	IsNull.	0.53
160.00	574990	FieldCopier.	0.28
531.00	2,249,938.00	Other	0.24
8601	2249938	Grand Total	3.82

Input is 25,000 records; each base operation may be invoked by one or more transformations, where "Other" represents time not used for invocation. The first column is the time total in milliseconds, the second column is the number of invocations. The 4th column is the average time per invocation. Due to clock resolution of 1 millisecond, the numbers are good only for relative weights. Yet, some of the base operations are indeed inefficient.

3.1.5 Java GC considerations

The third measurement looks at Java heap reallocation, or what's called "garbage collection". With a heap size of 0.5 GB, running the plan in up to hundred iterations, total heap allocations can be calculated, resulting in a usage rate of 130KB per 1 KB input record. GC sweeps all allocations for "dead" objects and frees this space, and possibly even doing space compaction. In our case, we found out that GC time occupies 25%-50% of total execution time. Still MCS providing the performance we need, yet much performance improvements are still possible via smarter object management and reuse.

3.2 The MCS orchestrated performance

3.2.1 MCS scalability

To achieve its performance goals, MCS are deployed as J2EE application on WAS. Specifically, MCS using WAS Network Deployment (ND - [2], [3], [4]) clustering capabilities provides MCS with the needed scalability with load balancing via messaging. With WAS-ND, MCS can be deployed on a number of machines, each serving as one or more nodes in the distributed system, each node managing one or more servers – each of which is a J2EE instance. Each server, in turn, manages a pool of threads on which the MCS processing entities execute their processing steps, when invoked by a trigger message as MDBs. The messaging engines in WAS (the System Integration Bus – or SIB) implements the Java Messaging Services (JMS) [15], and provides load balancing, ensuring that threads will work evenly distributed on the MCS allocated machines.

MCS maintains 4 different queues, each driving a different type of MCS processing entity. That is depicted in Figure 3-1, showing a plurality of Envelope and Task entities, according to the different input files flowing through MCS. The Poller is a system-wide singleton and its queue is used to simply start it up and periodically invoke it via timer messages (triggers) to rescan the input folder for new incoming files. The packages are also singletons, each created to handle one database loading destination.

WAS is configured to the number of listening points per server, per each queue – or number of concurrent MDBs. This dictates the level of parallelism on each server. Defining the number of threads is a critical performance issue, and is based on experience and tuning. Performance estimates as we will present in the next section are essential in doing that tuning. The general guidelines would be to keep CPU busy. MCS tests indicate the following thread distribution is a good start:

- Number of Task MDBs = number of CPUs + 20 %
- Number of Envelope MDBs = number of Package MDBs

- Number of Package MDBs = assumed number of assigned packages instances.
- Number of Poller MDBs = one.

The MCS data flow among its entities as well as their relations is illustrated in Figure 3-1.

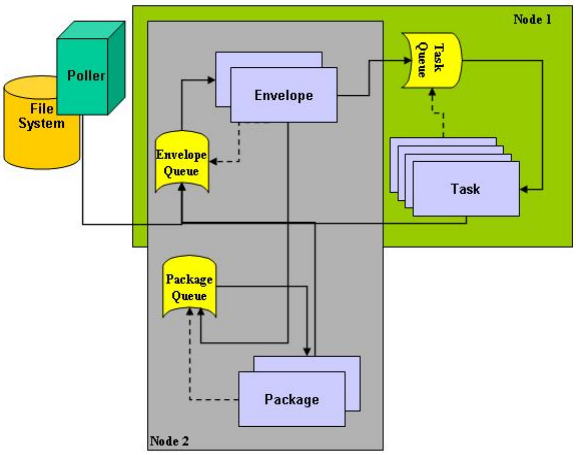

Figure 3-1: MCS Queue Data Flow

The above description is very brief, as we do not elaborate on the post task-processing in MCS which mainly includes database loading and other database processing. Our purpose in this paper is to discuss the massive data processing which occurs at the task level in a highly parallel way.

3.2.2 Linear model for mix of input

MCS input is a mixture of record types having different number of fields, fields types, processing plan (which drives different base operations), and different output fields and types. The incoming record rates for the different record types present what we call an MCS input mix. Looking at past results, one can create pretty reliable estimates for future and differing mixtures.

Although all the parameters controlling the MCs execution should be considered, we show that a very simple linear model works pretty well for gross estimates. It can be assumed that different MCS server configuration in which the number of MCS entities working in parallel is set up differently will impose different coefficients in that model.

To derive the estimation MCS is over loaded with a single record type, during which logs are collected indicating the start and end events of each transition and state action in a server. Using simple shell scripts these logs are processed to form an occurrence graph, as in Figure 3-2, with the occurrence number on the X axis, and event time on the *Y* axis. This graph show that each type of event is independent of the other events, and behaves almost linearly. Therefore, MCS behavior can be extrapolated linearly from a significant, but small sampling period.

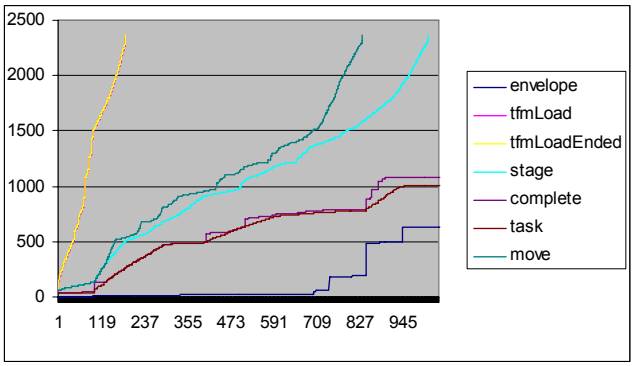

Figure 3-2: *Vanilla Record* Occurrence Graph
(1 Envelope MDB, 1 Package MDB, 8 Task MDBs).

By assuming a linear dependency of processing time on the size of input record, the MCS execution of records is drastically simplified – but shows pretty reliable predictions.

The MCS performance consists of two parts: processing input records and uploading the results to a database. That second part has not been dealt at depth in this report. Yet, we also measured for a certain uploading solution, the database loading time in average and made it depending on the output record size as well. Here too, we found a linear model to reflect nicely on the actual performance.

Table 3-1 shows a base plan we call "vanilla" working for a long period of time, processing input records at a rate of 340 million records a day, processing input records of size 535 bytes. Output records are of size 1359 bytes, and are loaded at a lower rate of 140 million records a day. That sets the overall system performance at the lower level, showing that it requires strengthening by allocating more computing resources to it.

Table 3-1. *Vanilla* Record Type Performance

MCS Plan	Input Record (bytes)	Output Record (bytes)	Performance (Millions of Record per day)
Vanilla	535		340
		1359	140

Let define n_j as the rate of processing records of type $j \in \{1,2,\ldots,k\}$, meaning the number of records processed at a given time frame (e.g., per day). Let s_j be the size of record j in bytes. Our model assumes inverse dependency of record size and rate, so that for $\forall i, j \in \{1,2,\cdots,k\}$, the following holds:

$$(3.1) \quad n_j * s_j = n_i * s_i = D$$

So, knowing n_0 (being our "vanilla" record type, all n_j can be estimated. Same as we do the above for input records, we can also

do for output records, estimating uploading rates of the output records into the database.

Given a distribution of records

$$(3.2) \quad D = (d_1, d_2, \ldots, d_k) \quad \sum_k d_m = 1$$

The overall estimated rate (e.g., number of records processed in a day) is calculated by the simple inner product:

$$(3.3) \quad N_T = \langle N, D \rangle$$

where

$$N = (n_1, n_2, \ldots, n_k)$$

The following tables compare estimates with actual performance results, according to the linear model of equations (3.1). Table 3-2 shows the estimation results for different record types applying equation (3.1) on the *vanilla* record as base. Table 3-3 shows distribution (aka equation (3.2)) of input record mix, and Table 3-3 shows the very good match between the actual results obtained in that test compared with the estimates.

Table 3-2. MCS estimations for different records types (million numbers of records per day)

Plan	Input (bytes)	Output (bytes)	Estimated Performance	
			Task Processes	DB Upload
Orange	3245	1565	56.1	125.9
Sugar	1914	1590	95.0	123.9
Rice	1123	1443	162.0	136.6
Water	455	886	399.8	222.4
Salt	246	644	739.4	306.0
Chocolate	246	659	739.4	299.0
Bananas	246	659	739.4	299.0

Table 3-3. MCS actual records distributions

Plan	Input (bytes)	Output (bytes)	Distribution (accounting failures)	
			Task Processes	DB Upload
Orange	3245	1565	2.80%	2.89%
Sugar	1914	1590	7.86%	5.68%
Rice	1123	1443	12.73%	12.35%
Water	455	886	0.62%	0.64%
Salt	246	644	1.83%	1.89%
Chocolate	246	659	49.11%	50.70%
Bananas	246	659	25.04%	25.85%
Total	--	--	100%	100%

Table 3-4. MCS total performance (million numbers of records per day)

Performance (MR/day)			
Task Processing		DB Upload	
Estimate	Actual	Estimate	Actual
594.03	620.89	263.66	276.16

4. Future work

MCS operates in a changing business environment. MCS deployments require a deep understanding of the underlying system resources needed to make efficiency forecasts affected by business changes, thus reducing risks in system operations and making it proactive to system operation hazards. To this end, the simplified MCS linear model needs to be further elaborated to include more coefficients and parameters which will make accurate estimations of processing times at all node as part of overall capacity analysis. MCS core is the relatively small set of base operations – so it is easy to analyze and parameterize for performance – unlike any other general purpose programming language. With MCS it is possible to identify emerging bottlenecks, and plot an operational course of action, either automatically, or as part of an what-if questions session with a system manager when considering cost/performance tradeoffs as well as other business factors when deploying MCS.

5. Summary

This paper describes a highly distributed massive collection system that processes a massive mix of input records according to different domain specific processing plans. The founding requirements consist of ease of use along with very high performance throughput rates. To this end, pure java implementation was determined as benchmark baseline for comparison. The MCS specification language together with its interpreter was shown to provide an efficient and adequate performance results compared to the pure java benchmark. On the massive highly parallel J2EE deployment on WAS, a simple linear model provides a reliable prediction to different input records distributions. Surprisingly, it seems that during massive load there is little dependency on the specific input records structure or processing plan, except for input and output records sizes.

6. ACKNOWLEDGMENTS

Our thanks to to Dagan Gilat, Pnina Vortman, and Yoel Arditi for their visionary drive in this project. To David Berk, Yaakov Dolgov, and Alex Akilov who contributed to the implementation and design of the solution.

7. REFERENCES

[1] Domain-Specific Languages: An Annotated Bibliography, Arie van Deursen, Paul Klint, Joost Visser, http://homepages.cwi.nl/~arie/papers/dslbib/.

[2] WebSphere Application Server V6 Scalability and Performance Handbook, IBM Redbook, SG24-6392-00, ISBN: 0738490601

[3] WebSphere Application Server V6 Technical Overview, by Carla Sadtler, IBM Redbook, REDP-3918-00

[4] WebSphere Application Server V6 Planning and Design WebSphere Handbook Series, IBM Redbook, SG24-6446-00, ISBN: 0738492183

[5] Java™ 2 Platform Enterprise Edition Specification, v1.4, Sun Microsystems®, Nov 2003, http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.

[6] Unified Modeling Language version 2.0 Specifications, Object Management Group, http://www.omg.org/technology/documents/formal/uml.htm

[7] The Eclipse project organization http://www.eclipse.org.

[8] "Contributing to Eclipse: Principles, Patterns, and Plugins," by Erich Gamma and Kent Beck, Addison-Wesley, 2003, ISBM 0-32-1205758

[9] The Eclipse Modelling Framework, http://www.eclipse.org/emf/emf.php

[10] "Eclipse Modelling Framework: a Developer's Guide," by Frank Budinsky et. al., Addison-Wesley, 2003, ISBN 0-13-142542-0.

[11] Extensible Markup Language (XML) 1.0, 3rd Edition - a technical recommendation standard of the W3C. http://www.w3.org/TR/REC-xml

[12] XML Metadata Interchange (XMI) Specifications, Version 2.0, May 2003, Object Management Group, http://www.omg.org/docs/formal/03-05-02.pdf

[13] The Awk Programming Language (Aho, Weinberg and Kernigham), http://www.uga.edu/~ucns/wsg/unix/awk/

[14] JDBC – Java DataBase Connectivity, API Specifications, Sun Develioper Network (SDN), http://java.sun.com/products/jdbc/download.html

[15] Java™ Message Service Specification Version 1.1, April 2002, Sun Microsystems®, http://java.sun.com/products/jms/docs.html