Why Source Code Dependency is a Challenge for Mainframe Workload Rehosting

Overview

The halcyon days of mainframe application development started in the 1970s and continued until the arrival of distributed computing in the early 1990s. It was during this period that most mainframe applications, which still form the backbone of over 70% of the world’s electronic commerce, were originally written. Few programmers who developed these applications thought their work would still be in use 50 years later; indeed, many of them didn’t think that the programs would even be in use in the year 2000.

Source code management in these pioneering days of commercial IT relied heavily on local system knowledge. The source code management disciplines we take for granted today only became part of the application development lifecycle as the job market for programmers exploded and staff turnover increased.

This loose relationship between source code, documentation, and the actual programs in use has been a developing problem for 50 years. And the difficulty in identifying precisely which source code was used to build aspects of long-running applications is central to some of the challenges of mainframe workload rehosting. As an IT executive in one large enterprise lamented, “we’ve got about 100 million lines of COBOL in our source code repository, but it’s OK as we only have 10 million lines active. We just don’t know which 10 million.” To consider recompiling such applications to run on a new platform requires the prohibitively daunting task of correctly identifying the source code and making it available for migration.

However, source code availability is far from the only issue at play when looking at rehosting mainframe workloads. Although a mainframe is fundamentally a Von Neumann architecture computer like any other, the way in which the operating environment was implemented, and has evolved over more than half a century, creates hardware and software dependencies that cannot simply be compiled away. Stated differently, even if the source-code is available, insurmountable problems remain.

This document looks at some of these non-obvious issues and the impact they will have
on any mainframe workload rehosting project based on source-code recompilation. It also explains how a Software Defined Mainframe can help to overcome these issues.

The Gordian Knot

When approaching a mainframe application rehosting exercise, many of today’s IT experts assume that mainframe applications benefit universally from the same level of inherent documentation as modern applications. Also, it is not widely understood that an application suite “runbook” is almost as important to understand as the application logic itself.

Specifically, it often comes as a surprise that an application dependency map is difficult to obtain. In this context, an application dependency map details all the files and programs that a program references or is referenced by. The challenge with mainframe workload rehosting is that, without this map, it is almost impossible to estimate ahead of time just how many programs and files will need to be rehosted to support the migration.

Once a user starts looking, it is very common for a list of program dependencies to grow exponentially. The initial program may reference two others, which may in turn reference another two, one of which may be called by several other programs. The files used by the first program may also be used by an entirely different set of programs, all of which have their own dependencies. In even a modest system, this Gordian knot of interconnections exponentially extends the scope and duration of the rehosting workload project beyond even the most pessimistic initial assessment.

When a rehosting strategy is based on recompilation of source-code, the situation becomes dramatically worse. For each program, the source code and any copybooks must be found — even code for the odd obscure utility that hasn’t been updated since 1979 (and they do exist in many mainframe application portfolios).

The Data Will Get You

Let’s assume for a moment that by some miracle every dependency is understood, and every program can be associated with its source-code and copybooks. The task is now reduced to a recompilation exercise. Well, not exactly...

As previously mentioned, mainframes, whilst fundamentally the same as all Von Neumann architecture computers, differ in some important ways when workload rehosting is considered. Because these issues are found in many different parts of a typical application, the amount of re-writing needed to migrate an application to a new platform can be daunting. Many of the most significant differences concern the way data is encoded or processed.

Issues with mainframe encoding

Mainframes store character data using encoding known as EBCDIC, whereas x86 computers, the most common target for rehosted mainframe applications, encode the data in ASCII or a more modern Unicode variant. While storing the data may be straightforward, the effect that the encoding has on program logic can cause serious complications.

Different encoding schemes have different sort orders (also called collating sequences). In ASCII, an ascending sort order puts numbers ahead of upper-case letters, ahead of lower-case letters. In EBCDIC, it’s the other way around. Sorting order affects many aspects of program processing, not least of which is the outcome of very common programmatic decisions points. For example, the outcome of the expression “If ‘a’ > ‘A’ THEN...” depends on the encoding of ‘a’ and ‘A’. You can see how a failure to take encoding into consideration can have a profound effect on the behaviour of the program being recompiled.

The result of this encoding issue is that the source code associated with programs being rehosted must be changed to reflect the sort-order differences between the encoding formats. Imagine the scale of this exercise, with the associated regression testing, if the application dependency map has identified hundreds of programs to be rehosted — a not unrealistic number.

Mainframes also maintain a unique representation of decimal data called Packed Decimal. Almost all applications that deal with money will have some binary decimal representation
somewhere. There is no equivalent of Packed Decimal in modern data encoding. Every copybook would need to be reviewed to see the packed decimal prevalence. While modern compilers for distributed systems can emulate packed decimal arithmetic, the challenges arise when the database system is expected to perform that arithmetic; more on this later.

Even integer numeric representations, that appear to be the same across systems, are in fact very different at the bit level. Mainframes store multi-byte values in big-endian order whereas x86 systems store them in little-endian order. The impact of this difference in encoding is felt when data is moved “as is” to x86 and then subsequently processed. Integers maybe syntactically valid, but their values are totally changed.

**Issues with data processing and databases**

Encoding issues are just the tip of the iceberg. Differences in data processing and database implementations also introduce problems in application rehosting.

Modern computers perform floating-point operations differently than mainframes; the maths yield a different result. For certain classes of applications, failure to account for these differences can have a very serious impact.

Mainframe databases are implemented differently from those of other systems. For example, there is the issue of SQL, which is a “recognised” standard, is incompletely specified and as such leaves room for “creativity” across implementations. Not only is the mainframe dialect of SQL (based on the DB2® database system) different from non-mainframe variants, but mainframe applications frequently use embedded static SQL, which is uncommon on other platforms. Static SQL was implemented early in the life of DB2. It was developed to enable SQL to perform adequately when used in support of complex queries.

Relational databases have a component known as an optimizer to determine the most efficient way to retrieve data from multiple relational tables; known as an access path. In the early days of DB2 the optimizer itself took a long time to evaluate the best access path. Static SQL was introduced to enable the optimizer to run offline, with the resulting access path pre-calculated for the SQL in the program. Indeed, the program itself no longer needed to have the SQL statement with the code. The program would simply refer to an object in the DB2 catalog to run the SQL statement. This creates a big problem when recompiling mainframe source code, with static-SQL references, to run against a non-mainframe database with no concept of static SQL. Once again, significant code changes need to be made to convert embedded static SQL into dynamic SQL or separately stored procedures and adapt the program flow to support the adapted referencing of the database system.

Also, we have the issue of mainframe datatypes within non-mainframe database systems. Although many mainframe datatypes are common in other systems, two of the most prevalent datatypes, character and decimal, are different.

Character columns in a relational database have all the same issues of collation that exist in the native file system. Unless the database is aware of these issues and performs the same way, the program will need to be changed to mask these differences.

Database systems often perform arithmetic on numeric fields before returning the result set to the program. When those fields are packed decimal, unless the database system has support for both packed decimal fields, and their associated arithmetic operations, then the source code for the program will have to be changed to compensate for this.

The transactional behaviour of the different relational implementations is also different. In some databases, a failing SQL statement results in the unit of work being aborted, yet other implementations (like those on legacy mainframes) allow the application to take corrective action and continue. Many customer programs are dependent on this type of implicit behaviour, something which cannot be easily deduced from the source code.
Expect the Unexpected

Another rarely anticipated aspect of mainframe workload rehosting is the challenge presented by the use of obscure coding techniques that are idiosyncratic to the mainframe environment. The excerpt of code below is an example of a mainframe-specific implementation that may have been clever in its time, but would require rewriting before it could be compiled to run on a different platform.

This COBOL program excerpt follows chains of z/OS® (the mainframe operating system) control blocks to find the name of an object required by other parts of the application. Because many control blocks are linked together in a well-understood manner in the mainframe environment, such techniques have been employed by programmers to circumvent operating system shortcomings. However, these arcane routines become impossible to maintain once the original programmers leave the company. Orphan code such as this is a great example of the legacy code problem—it creates an almost insurmountable lock-in to a mainframe operating environment. Because the program behaviour depends on system level features of the mainframe environment that are not found on other platforms, there is absolutely zero chance such a program could be simply compiled to work on another base architecture.

How a Software Defined Mainframe Solves These Issues

By providing an application hosting environment that is fully interoperable with the legacy platform, a Software Defined Mainframe turns this previously impossible application rehosting challenge into a manageable task.

Overview

LzLabs Software Defined Mainframe® provides functionality that enables binary programs, originally compiled for Mainframe Instruction Set Architectures, to run without any changes or recompilation on x86 Instruction Set Architecture computers. Stated differently, there is no requirement to locate the source code, change it to ensure the logic will not be affected by recompilation, recompile and run extensive regression tests.

To operate without requiring the source code, a Software Defined Mainframe must, from the point of view of the migrated legacy application, be interoperable with, and fully support the functionality of, the legacy mainframe platform. This does not mean that it is a software replica of a hardware mainframe; far from it. What it means is that the program calls to the language environment, I/O, databases, online, batch, and security subsystem interfaces have functionally-equivalent behaviours to those found in their counterparts on a legacy hardware mainframe.

Programs

A Software Defined Mainframe is capable of supporting mainframe-specific features that enable coding patterns unique to mainframes. Without such features, any application program that chains through control blocks, or exploits other platform-specific capabilities, would have to be re-written—and in some cases, may simply be impossible to rehost.

Other examples of operating environment programming patterns including certain VSAM file access techniques. It is not uncommon for certain early programs to access VSAM at the control interval (Ci) level rather than record level. Cis are low-level system structures and were used in the early days to optimize storage and performance. Like the chained control blocks, Cis have no equivalent outside the mainframe environment. As a result, recompiling such programs to run on a new platform will require rewriting and extensive testing.

As these examples show, laying hands on the source code, challenging as that is, solves only part of the problem. To recompile many mainframe applications to run successfully on the target environment, one must substantially change the code, then compile, and conduct extensive validation and testing to assure identical results.
Similarly, for programs that perform low-level, CI-based access to VSAM, the Software Defined Mainframe supports such behaviour using an API that enables such accesses without rewriting the underlying code. In addition, the data is stored in such a way that VSAM behaviours, such as CI splits, will operate normally from the perspective of the program.

As these examples show, the Software Defined Mainframe makes application rehosting possible by supporting the mainframe-specific features available to applications. Without this support, the program migration would require time-consuming and expensive rewrites, recompilation and verification.

Data

By storing the data in its native format, and implementing sorting based on that format, the collation and number representation issues identified earlier in this paper are eliminated.

An organization using a Software Defined Mainframe can be sure the rehosted program logic will have been unaltered and will produce identical results, even to the very digit of high-precision numbers (COBOL allows numbers with 31 significant digits).

Furthermore, the LzLabs Software Defined Mainframe® has an SQL intercept layer that faithfully recreates the behaviour of static SQL from the perspective of the program. By enabling static requests to run queries, the need to modify, and therefore find the source code for, programs that reference DB2 is eliminated. This layer, known as LzRelational™, builds on top of PostgreSQL. PostgreSQL is one of the most widely used Open Source relational databases available today and supports many of the world’s most demanding relational workloads.

LzLabs has also exploited the PostgreSQL User-Defined Datatypes mechanism to implement those DB2 datatypes not found in non-mainframe relational databases. This addition to PostgreSQL further ensures that program behaviour is consistent across platforms without having to make code changes.

These user-defined datatypes also possess functions that behave the same way as their mainframe counterparts. For example, LzLabs has user-defined functions for Packed Decimal columns, which perform packed decimal arithmetic on those fields. This functionality ensures that no application changes are required for this extremely prevalent legacy application functionality.

Crucially, the use of user-defined datatypes opens the door to greater flexibility with legacy data. Due to the way LzLabs has implemented its user-defined datatypes, the data can not only be accessed in its legacy, mainframe format, but also may be accessed in standard Linux formats, such as ASCII/Unicode. This new flexibility unlocks access to legacy data for new applications; for example, programs written in Java and accessing the data using standard JDBC. The master copy of the data is liberated from mainframe limitations for new uses (think here real-time analytics and Big Data) while remaining fully accessible to legacy application for business continuity purposes.

A Final Word About Source Code

In some cases, there is no escaping the need to work with the legacy source code. Many mainframe applications are regularly updated for business or regulatory changes. Even though the source-code of such applications is, due to the regular updates, known and available, recompiling it to run on an x86 platform remains a challenge. Code changes to support new business demands are one thing. Code changes to support rehosting are far more invasive and unpredictable.

LzLabs has developed a compiler layer that enables changes to be made to these “actively maintained” legacy applications to support changing business needs while low-level, mainframe-specific aspects of the legacy system are left untouched. Because the compiler layer targets the Software Defined Mainframe, mainframe-specific characteristics of the application are retained. For example, if the program was originally written to exploit EBCDIC data encoding and static SQL, those programming patterns will be preserved even though the program has been recompiled using open systems compiler technology.

Furthermore, the LzLabs development infrastructure for these active programs provides many of the popular features and tools used in modern development environments. Remote debugging, IntelliSense®, and syntax highlighting all combine to make the ongoing maintenance of active legacy programs far more accessible to the current generation of programmers.
Summary

Mainframe workload rehosting is increasingly popular. But the traditional approach of requiring the recompilation of all source code is fraught with issues — issues that are not apparent at the whiteboard stage of any project. The arcane legacy of 50 years of mainframe programming, has created a bewildering array of barriers to success for any project that relies on source code – barriers made insurmountable by the skills shortage at large within many users of legacy hardware mainframes. A Software Defined Mainframe is the only plausible option for organizations wishing to rehost and modernize. It provides a low-risk, practical solution that serves as an excellent platform to enable businesses to continue to use their substantial investments in legacy application software as part of modern digital strategies.