USENIX Association

Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

Migrating an MVS Mainframe Application to a PC

Glenn S. Fowler, Andrew G. Hume, David G. Korn and Kiem-Phong Vo
AT&T Labs – Research
180 Park Avenue, Florham Park, NJ 07932, USA
{gsf,andrew,dgk,kpv}@research.att.com

Abstract

Due to advances in computer architecture, performance of the PC now exceeds that of the typical mainframe. However, computing cost on a mainframe continues to greatly exceed that on a PC. Thus, migrating mainframe applications to PC can result in substantial savings. The major stumbling block in doing this is the cost of software migration itself. This paper discusses an experiment in using a software tool approach to migrate a large billing application from MVS running on a mainframe to UNIX running on a PC. We developed tools to port the application from mainframe to PC with minimal code rewriting and enable transferring data cheaply so that processing can be done without interrupting other ongoing mainframe operations. We were able to transfer data from the MVS mainframe to a Linux PC and complete its processing in less total time than if done entirely on the original mainframe.

1 Introduction

Approximately 25 years ago John Linderman at Bell Laboratories wrote a Technical Memorandum describing the UNIX *sort* program. The state of Unix computers was such that John concluded that it was faster to sort a file by writing it to tape, transferring the tape to the mainframe to sort, and then loading the result back to the Unix machine. Much has changed in 25 years. Processor speed, memory, disk capacity, and network speeds have followed Moores law and improved exponentially. A 3GHz processor is cheaper now than a 1Mhz processor was then. A gigabyte of memory costs around $300 compared to $30,000 for a megabyte. A 250 gigabyte disk now costs around $250 compared to about $10,000 for a 100 megabyte drive. Meanwhile, networking speeds of 64 kilobits a second have increased to a Gigabit a second. The price/performance of mainframes has lagged behind the PC's to the extent that a mainframe computer with the equivalent power as a PC now costs at least two orders of magnitude higher.

Beyond hardware costs, the software costs for mainframes are also at least an order of magnitude higher than that for PCs. There are two reasons for this. First, the number of mainframes is substantially less than the number of small computers so that software production costs are amortized against a smaller population. Second and more importantly, software development on mainframes now requires specialized, vanishing skills due to the use of antiquated programming languages and tools such as COBOL and JCL. To contain the high cost of software development on mainframes, Information Technology organizations have experimented with various approaches to migrate computing to smaller computers.

The most direct approach to migration is to simply rewrite mainframe applications to run on PCs. This task is difficult as everything from the operating system, databases, the languages, and even the character set representations are different between these computing environments. Further, the people who wrote the original system may no longer be around and there may not be enough documentation to provide a complete description to duplicate. This means that massive code reengineering effort is necessary. Lastly, even if a large software application is successfully rewritten, it may still need to interoperate with other mainframe systems. This means that data migration and/or transcoding must be addressed upfront along with code rewriting and scalable data transport between machines must be available. Although many software tools and techniques for software and data reengineering [2, 6, 7, 15] are available, the risk in conducting such a migration effort is enormous. The peril of this approach was highlighted recently in an article on the New York Times * on the cost overrun in an effort to modernize the Internal Revenue Service software system.

A more reasonable approach adopted by a number of IT organizations is to encapsulate the mainframe by providing standard access to data, and then using PC's to add new functionality. While this strategy does make it possible to add new features more cheaply, the large central costs of maintaining the mainframe remain. In addi-

* http://www.nytimes.com/2003/12/11/business/11irs.html

tion, as the data streams between the different environments diverge, it becomes difficult to perform decision supporting tasks such as data mining that may require fine-grained data integration and manipulation.

The strategy which we chose to explore is to develop tools that allow much of the current mainframe software to run on the PC with little or no change. The original code can be either automatically converted or interpreted. As the software must handle both mainframe and native PC data, tools are provided to transparently and cheaply move data between the two environments. Of course, the software on the new system still requires much of the same skill set to maintain as it did on the mainframe. However, once on the PC, the software can be incrementally rewritten. In this way, significant savings can be realized early with small upfront costs. There have been a few efforts along this direction, most notably the works by Henault [12], Rossen [17] and Townsends [18]. Both of these stopped at translating the Job Control Language scripts into Unix shell scripts.

The rest of the paper describes the tools and techniques developed in an experiment to move a mainframe application to PC's. The ported application was run alongside its mainframe original to test correctness and measure performance.

2 The application to be migrated

Daily operations in a corporation like AT&T are dominated by large mainframe applications. It is important for the success of any migration project to quickly show advantage over existing practice. This implies the below criteria for selecting a test case for our approach to software migration:

- *Ease of implementation*: For fast demonstration of concepts, we wanted an application that would showcase the use of tools. In particular, our AST Toolkit [9, 10] provided a large collection of software tools for various aspects of computing, ranging from compression [21] and sorting to scripting [13] and software configuration [8]. So we wanted an application making extensive use of these techniques.

- *Correctness and significance*: To show effectiveness, we wanted an application with sufficient processing complexity, dealing with large data and independent from any MVS database. The database independence aspect enabled running the migrated version on the PC alongside its mainframe counterpart and testing the results. At the same time, cost savings could be shown by measuring the performance of both versions.

The application that we chose was a part of the VTNS Billing Edge biller. This system processed billing records for approximately 340 business customers per month. Billing records were fixed size, 650 bytes, but the number of records per customer could vary anywhere between one and around 40 million. Data processing was done in two cycles. The larger cycle on the 10^{th} processed about 290 customers with total data about 560 gigabytes and required about 60 CPU hours of mainframe computing. The smaller cycle on the 31^{st} handled the rest of the customers with approximately 140 gigabytes of data and required about 24 CPU hours.

Table 1: Code sizes

Language	#Files	#LOCs
COBOL	30	15K
JCL	900	100K

Table 1 summarizes the size of the application. COBOL is the main programming language used on mainframes. JCL is the Job Control Language used to write scripts to execute processes similarly to the shell language [13] on a Unix system. We note that the chosen application is just a small part of the entire biller which is more than 1.5M COBOL LOCs.

Figure 1 shows the data processing for this application. The names of the components are as defined by the overall biller. The workflow among the components is as follows:

- The input data to our application is generated by three external systems encapsulated in the diamond boxes.

- The JCL scripts, `VMURH1` and `VMURH4` (in ellipses), run COBOL programs to process this data into records sorted in various ways. Certain non-key fields of records that compare equal may be summarized to produce various sorted output files. Note also that `VMURH1` is generic and only driven by customer-specific data but `VMURH4` is a JCL script generated per customer.

- The output of `VMURH1` for all customers are then merged together by JCL scripts `VMURH31Q` and `VMURH31!Q` into files by types.

- Finally, the files grouped by types via `VMURH31Q` are processed by a set of ten processes `VTUDHR[01-10]` into files to be processed by other systems.

The internal working of the above software components are opaque to us. However, understanding them is

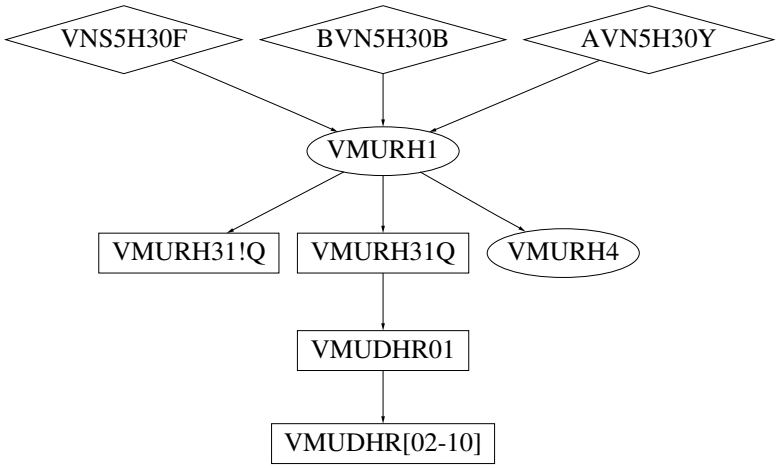

Figure 1: Processing customer data.

not necessary in a tool approach to migration. We only need to ensure that the processes can be compiled and executed or interpreted on the PC's to produce the same data as on the mainframe.

3 Software tools

Our AST Toolkit consists of an extensive collection of tools portable across nearly all flavors of Unix. In fact, these tools run transparently on both the PC and the UNIX System Services [1] part of MVS. However, there are major differences between mainframe MVS and PC UNIX that necessitate a number of new tools and techniques:

- Software to copy files between mainframe and PC.
- A COBOL compiler.
- A way to read and execute the MVS job control language, JCL.
- A sort program compatible with the IBM sort program.
- A way to schedule jobs on the PC's to efficiently process data.

3.1 Copying data between mainframe and PC

There are number of issues in moving data between MVS and PC. We discuss each below:

- *File system differences:* Much of MVS data are stored in MVS partitioned data sets. MVS partitioned data sets are similar to UNIX archives which can be mapped into Unix directories on the PC. Fortunately, the MVS Unix System Services provides a *cp* command that can extract the individual files from a partitioned data set and copy them from the MVS file system to the UNIX file system. In particular, it copies a partitioned dataset into a Unix directory so that each member becomes a file. Then, the *pax* utility can be used to archive this directory into either *cpio* or *tar* format and transfer the data to the PC via *ftp*.

  COBOL source files, however, are not stored in a partitioned data sets. Instead, they were stored in a sequential data set in a format understood by an MVS tool named *ca-librarian* [3]. As the interface to *ca-librarian* is interactive and menu-based, it is difficult to extract and copy multiple files in bulk. We reengineered this data format and built it into our AST *pax* command. This allows us to simply copy the *ca-librarian* sequential data set and read it on the PC using *pax*.

- *Expensive data movement:* A major problem with moving data between the mainframe and the PC is volume. With existing data connection, transfer rate is between 1 and 2 megabytes per second. A month worth of data for our application is about 700 gigabytes. This means that transferring data from mainframe to PC alone could be anywhere between 97 hours up to 184 hours. Thus, we investigated compression tools such as *gzip* and *compress* to reduce the data size. As observed elsewhere [4, 5, 19], fixed-length records data were amenable to compression by first transforming the data to be column-major instead of row-major. We wrote a simple compressor that transposed rows and columns, then applied run length and static Huffman encoding.

This technique compressed three times better than *compress* and twice better than *gzip* and ran about 4 times faster than both tools. On MVS, we were able to compress a gigabyte of data in 80 seconds and got about a factor of 25 compression. In this way, the 700 gigabytes per month of data could be compressed and transferred in around 20 hours.

- *Different character sets:* MVS uses EBCDIC to store text while Unix uses ASCII. This means that text files must be converted to ASCII en route to the PC and back to EBCDIC in the other direction. Since our compressor was built into the Vcodex package [21] which supports general data transformation, it was a simple matter to add character set transcoding before encoding or after decoding.

3.2 COBOL

With about 15 KLOCs of COBOL to be migrated, even if we were proficient in COBOL (and we were not), it would have taken a significant amount of time to rewrite the application in C. Thus, we looked into obtaining a COBOL compiler for PCs. After failing to get proper licensing terms for a commercial COBOL compiler, we started experimenting with openCOBOL, an Open Source COBOL compiler written by Keisuke Nishida [16]. Although this compiler did provide most needed features, there were a number of changes needed to support a large application like ours. We discuss these next:

- *Language additions:* The openCOBOL language lacked a number of features supported by the IBM COBOL compiler. For example, IBM COBOL provides an `ENTRY` statement that enables multiple entries to a procedure and a `WHEN-COMPILED` preset variable stores that date and time that the program is compiled. These features had to be added in order to compile the existing COBOL code.

  The IBM compiler allowed specifications such as `ASSIGN TO DA-S-VMUR102B` without quoting whereas the openCOBOL one required that the dataset name be quoted. We modified the compiler to accept this syntax and to skip over the `DA-S-` prefix and then use the `VMUR102B` as the name of an environment variable to search for to find the actual file name.

  With JCL, multiple files can be specified for each dataset. In that case, the files are virtually concatenated into a single input data stream. We modified the openCOBOL compiler to accept a space separated list of file names as the value of an environment variable and open them sequentially as if they formed a single concatenated file.

  The generation of the `main` program was modified so that a `USING` clause would get the data from the first argument passed to the program.

- *Performance:* Initial testing of the compiler revealed that it could not handle files larger than 2 gigabytes. After fixing this problem, we found the compiler to be too slow mostly due to its arithmetic processing. We were able to rewrite this part of the code and some other parts to improve the speed of compiled code by a factor of five. Adjusting for the relative speed of computation on MVS, the compiler now produces faster code than that produced by the IBM compiler.

- *Character sets*: Since the compiler is ASCII-based, string data in data files must be in ASCII. We started by converting string fields in each record from EBCDIC into ASCII and left numeric fields alone. This was not sufficient since these data files may have string and numeric data in the same columns across different records. The solution was to convert all bytes to ASCII and modify the arithmetic conversion routines of the compiler to convert back to EBCDIC when doing the conversions.

- *Processing compressed data*: We modified the compiler to automatically handle data compressed by the mentioned compressor. By convention, such compressed data are kept in files whose names end in `.qz`. Thus, if the compiler opens such a file for sequential reading, data is automatically decompressed before reading. Conversely, for sequential writing, the data is automatically compressed.

We were able to work with Nishida to add all of the needed language features as well as other modifications into the openCOBOL compiler. In this way, future projects migrating COBOL programs can build on our work.

Finally, the compiler converts each COBOL module to C and then invokes the GNU C compiler, *gcc*, to build an object file. The object files are then linked with a runtime library supporting various COBOL features to make an executable program. We wrote *nmake* makefiles [8] to automate this process. However, given the large number of makefiles that must be written in a large application, we are investigating automatically generating them.

3.3 Sorting

The MVS *sort* program has a number of features beyond normal sorting. For example as records are read, files can be created by selecting certain subset of the fields to create new records. Records comparing equal by keys may

also be merged by summing certain field values. The description of what keys to sort on, how to do merging, and what additional files to create is defined in a separate specification called *sort control cards*. These features of MVS *sort* were used extensively in our chosen application.

The AST *sort* program is a superset of the UNIX *sort* utility defined in the POSIX standard. A feature of the AST *sort* not apparent at the command level is that it is just a driver on top of a sorting library designed in the *Disciplines and Methods* paradigm [20]. This paradigm provides a standard API via the *discipline* mechanism to extend library functionality. We were able to duplicate the required functionality of MVS *sort* by writing a few disciplines that make use of MVS sort control cards to process a record when it is read or to merge records compared equal in the same way that MVS *sort* does it. The *sort* disciplines are implemented as shared library plugins. This means that discipline-specific overhead is only incurred after the plugin is loaded at runtime.

For full MVS compatibility, we extended AST *sort* to deal with fixed length records and binary coded decimal fields. Similar to modifications to the COBOL compiler, the sort program could also handle concatenation of files and data compression with the `.qz` suffix.

On MVS, our *sort* runs about 5% faster than MVS *sort*. However, the two *sort* programs occasionally produces records in different order since ours is stable while MVS *sort* is not. That is, our *sort* preserves file order for records that compare equal by keys while MVS *sort* does not provide this guarantee.

3.4 JCL

JCL, the job control language for MVS, plays much the same role as the UNIX shell does in that it invokes programs or scripts in some order and takes actions based on the results. The chosen application executes over 100 thousand lines of JCL about 90% of which are generated and the remaining 10% are fixed. A JCL script is generated for each customer by accessing the DB2 database. These scripts merely call MVS *sort* with various control card decks generated from within the scripts. To eliminate these JCL scripts would require access to the database from UNIX. To avoid this complexity, we kept the generation of these scripts on MVS and then processed them on UNIX.

We wrote *jcl*, a JCL interpreter, that allows the use of the hierarchical Unix file system instead of the flat MVS file names via file name prefix mapping. For example, a partitioned data set for control cards, say `SYS1.CTLCDLIB`, would be mapped to the directory `${BILLROOT}/cntlcard` so that its control cards would be stored in separate files in this directory. *jcl* first parses JCL scripts into a linked list of program step structures. This list is traversed to either generate *ksh* shell scripts or to execute. *jcl* also provides debugging support that can be used to determine the overall structure and relationships between a collection of JCL scripts. For example, the `--noexec` option interprets the JCL but does not execute external programs and the `--list=`*item* option lists the *items* referenced by each JCL step.

3.5 Job scheduling

An application on MVS consists of a number of jobs some of which can run in parallel while others must wait until some other set of jobs finish before they can start. Let A and B be two jobs. We say that there is a directed edge $A \rightarrow B$ if there is a constraint that A must be completed before B can start. In this way, the set of jobs and constraints form a directed graph called the *scheduling graph*.

Figure 2 shows a slice of the scheduling graph for our application based on the processing of just two customers *xx* and *yy*. This essentially executes the workflow presented in Figure 1 except that all customers are now being considered together so there are more opportunities for parallelization. For example, as soon as the data produced by the external systems AVN5H30Y, VNS5H30F and BVN5H30B for a particular customer are available, the VMURH1 process for that customer can be started. As long as there are enough processing power, the scheduler starts many such processes in parallel. The results from these processes are further processed. In particular, the merged results by VMURH31Q are passed on to the VTUDHR processes.

By necessity, a scheduling graph must be acyclic so that the jobs can be scheduled. In general, jobs may have attributes associated with them such as completion time or memory and disk resource constraints. In that case, it is desirable to compute a schedule that optimizes some parameters based on these attributes. When only a single processor is available, any topological sort ordering of the jobs produces a valid optimal schedule. However, on a system with multiple processors, the scheduling problem is known to be NP-hard[11].

The MVS scheduler, New-Dimension, allows scheduling constraints to be specified by filling out form tables. Then, the scheduler controls resources of the system and sequences the jobs appropriately. For the PC's, we opted to write a simple scheduler, which reads lines from one or more queues specified as files, and runs a command for each line it reads. At startup, the scheduler is given a list of process resources and will run a single job at a time through each resource. If all resources are in use, the scheduler blocks until one becomes available. If all the file queues are empty, the

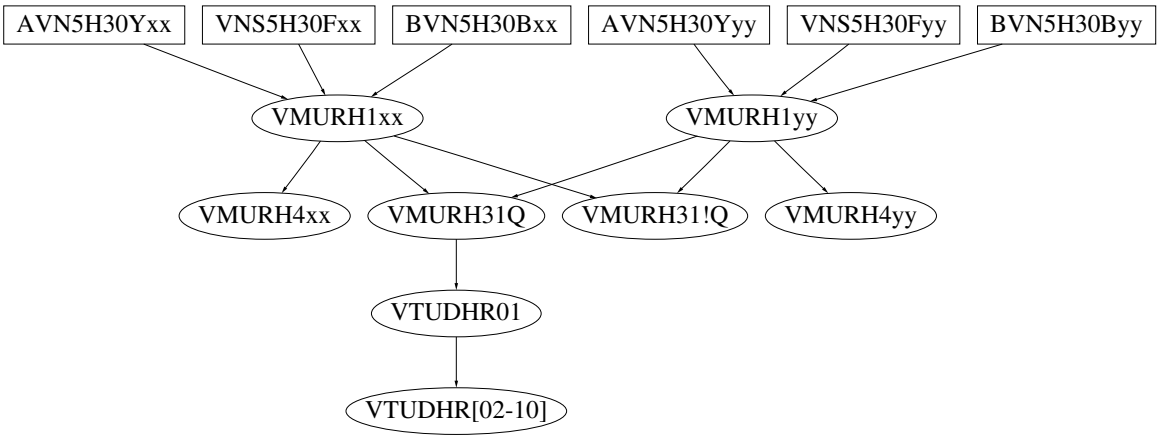

Figure 2: A scheduling graph.

scheduler will block until one of the file queues contains some input to process.

The simple scheduler is adequate for our prototype. However, in general, MVS scheduling has many facets that we did not account for. We are investigating writing a tool to read the MVS scheduling tables and perform the appropriate scheduling on the PC's.

4 Selecting a platform

We wanted to select a platform of hardware and software fast enough to do the processing, able to store the data, able to run all of the software, and at as low cost as possible. Based on price performance considerations, we built a cluster of two machines, each with a 2.8 MHz. Intel Pentium 4 processor, 1 gigabyte of 400 MHZ DDR RAM, and two 256 gigabyte SATA disk drives. The machines are networked with Gigabit Ethernet. Unit testing various CPU intensive programs showed that these machines were about 7 times as fast as the MVS systems. The SATA disk drives were able to tranfer about 100 megabytes a second. The entire cost for both machines was under $4,000.

Since most of the processing was per customer and could be done in parallel, we kept the machines loosely coupled with separate file systems instead of using a single shared file system. As data processing was intensive, avoiding the overhead of a shared file system such as NFS was a win overall. In data processing phases where files must be merged, they could be copied to wherever needed. Keeping a loosely coupled cluster of machines made it simple to disconnect a bad machine or to add new machines as needed. This increased fault-tolerance and scalability.

The choice of an operating system was constrained by the software to be run. The use of the AST Toolkit did not constrain this choice in any way. However, the openCOBOL compiler required the GNU-C compiler *gcc* and certain GNU libraries. We finally chose Redhat Linux 9.0 primarily because we felt it would be easier to integrate our solution into the billing application supported by IBM. Our concerns with Redhat Linux were primarily its erratic I/O performance. For example, the throughput was actually higher by running a single customer at a time on each system rather than running multiple customers in parallel. However, we were not locked into any operating system. We experimented running the software with FreeBSD Unix. In contrast to Linux, the throughput did increase when running two customers in parallel on FreeBSD Unix. Finally, a UNIX based solution was chosen rather than Windows mostly for reliability consideration. However, all the tools could be run on Windows using the UWIN software [14].

5 Results

Table 2: October 10, 2003 cycle.

Data	#Files	Raw	Comp.
From mainframe	300	560GB	22GB
Generated on PC	1160	280GB	17GB

For testing, we processed data for the October 10, 2003 cycle. Table 2 summarizes the data that were transferred from the mainframe and generated on the PC's. Theoretically, the data could be compressed on the mainframe in about 12.5 hours and transferred to the PC's in about 6 hours for a total of 18.5 hours. However, it took us over 24 hours to move the data from the mainframe to the PC's, mostly due to long waiting time for mainframe

tape drives. Once the data arrived on the PC's, processing completed in under 19 hours. Thus, even with the slow data transferring time, the output data was generated in about 43 hours. The same job took a total of 60 hour processing time on the mainframe. Note that, in this case, we waited until all files were copied to the PC's before processing. Our scheduler could be modified to allow overlapping of data transferring and data processing to reduce elapse time even further. In addition, our set up could be easily scaled up by adding more PC's. We estimated that with a 4 processor system, we could process this data in under 12 hours.

CPU cycles on the mainframe costs about $20/hour and network charge for data transmission to and from mainframe is around $5/Gbyte. Thus, if the above data is representative of the 10^{th} cycle, it could be compressed and transferred between the mainframe and PC's for less than $1000. Generously doubling this to cover both the 10^{th} and the end of the month cycles, the total cost for data transfer each month would be less than $2000. Assuming that the PC's cost $2000/month to own and operate, the total cost for migrating to the PC's would be under $4000/month. As the current cost to run the application on the MVS mainframe is estimated to be about $20000/month, the data processing cost can be reduced by more than a factor of 5 for this application using our approach.

6 Conclusions

We presented a methodology to migrate mainframe applications to PC's based on software tools. This approach minimized software and data reengineering and enabled smooth transition between the mainframe and the PC's. The work took place over a six month period and was carried out by a small team of software experts without prior MVS or COBOL skills. Much of the effort was spent in learning MVS, COBOL, and in writing reusable tools. Thus, the work could be easily duplicated on more ambitious problems with far bigger payoffs.

In an experiment using the developed tools to move a small but significant mainframe application to a system of two PC's costing less than $4000, we showed that over 80% of the monthly computing cost could be reduced, saving more than $16000 per month. This cost improvement was conservative. Most of the cost was driven by moving data from the mainframe to the PC's and that could be easily eliminated by getting the data directly to the PC's. The migration of the entire billing application to PC's might save up to 90% of the ongoing costs.

Beyond software migration, the work on compression and sorting were of a general nature. For example, the table compressor could be used to compress any database tables using fixed length records. It was shown elsewhere [19] that mainframe data of the type mentioned here could be compressed by factors anywhere between 50 to 100 to 1. Thus, the compression tool could be used to save disk space and tape usage on the mainframe. Our *sort* tool employs better sorting algorithms than MVS *sort* and the commercial *Syncsort* (http://www.syncsort.com/) software used in original project. Thus, it can be used to both improve processing and save the rather steep licensing fees being paid yearly.

Looking forward, there are a number of threads to be developed. For example, our test case does not involve database and report generation issues which are important in large applications. The questions of reliability, effective scheduling of processes, and optimal partitioning and placement of large data on a cluster of loosely coupled PC's are of independent interest. Such problems should be dealt with in conjunction with migrating a larger and more comprehensive application. We are looking into that.

Acknowledgements

We would like to thank a number of colleagues who provided assistance in this project. Silvia Coble and Robert Cummins suggested the application to migrate and helped obtaining data to understand what was needed. Jim LaBarge from IBM helped with MVS accounts and the Unix System Services. Doug Blewett from AT&T Research helped select the hardware platform, assembled it, and got it running. Finally, Chris Olsen administered our system.

References

[1] z/OS UNIX System Services. In *http://www.ibm.com/servers/eserver/zseries/zos/unix/*.

[2] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[3] Computer Associates. CA-Librarian User Guide 4.3 for OS/390, z/OS, and VSE. In *http://www.dcs.rochester.edu/Documentation/CA-Librarian%20V4.3/lib43%20User%20Guide.pdf*, 2001.

[4] A. Buchsbaum, D. Caldwell, K. Church, G.S. Fowler, and S. Muthukrishnan. Engineering the Compression of Massive Tables: An Experimental Approach. *Proc. 11th ACM-SIAM Symp. on Disc. Alg.*, pages 175–184, 2000.

[5] A. Buchsbaum, G.S. Fowler, and R. Giancarlo. Improving Table Compression with Combinatorial Optimization. *Proc. 13th ACM-SIAM Symp. on Disc. Alg.*, pages 213–222, 2002.

[6] Y.-F. Chen. The C Program Database and Its Applications. In *Proc. of the Summer 1989 USENIX Conference*, pages 157–171, June 1989.

[7] Y.-F. Chen, G.S. Fowler, E. Koutsofios, and R.S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance*, 1995.

[8] Glenn S. Fowler. The Fourth Generation Make. In *Proc. of the USENIX 1985 Summer Conference*, June 1985.

[9] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for Writing Reusable Libraries. In *Proc. of the ACM SIGSOFT Symposium on Software Reusability*, pages 150–160. ACM Press, 1995.

[10] G.S. Fowler, D.G. Korn, S.C. North, and K.-P. Vo. The AT&T AST OpenSource Software Collection. In *Proc. of the Summer 2000 Usenix Conference*. USENIX, 2000.

[11] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing ans scheduling: a survey. In *Ann. of Disc. Math.*, pages 5:287–326, 1979.

[12] H. Henault. A.C.M.U. : Automate de Conversion MVS-UNIX. In *http://www.hhns.fr/products/acmu/acmudoc.html*.

[13] David G. Korn. ksh: An Extensible High Level Language. In *Proc. of the Usenix VHLL Conference*, 1994.

[14] David G. Korn. Porting UNIX to Windows NT. In *Proc. of the 1997 Usenix Conference*. USENIX, 1997.

[15] H.A. Muller, J.H. Jahnke, D.B. Smith, M.D. Storey, S.R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE — Future of SE Track*, pages 47–60, 2000.

[16] K. Nishida. OpenCOBOL Home Page. In *http://www.opencobol.org*, 2003.

[17] E. Rossen. The MVS to UNIX migration HOWTO. In *http://people.linux-gull.ch/rossen/software/migration/migration.html*.

[18] O. Townsends. The Vancouver Utilities for Unix and Linux. In *http://www.uvsoftware.ca*.

[19] B.D. Vo and K.-P. Vo. Using Column Dependencies to Compress Tables. Data Compression Conference, 2004.

[20] Kiem-Phong Vo. The discipline and method architecture for reusable libraries. *Software—Practice and Experience*, 30:107–128, 2000.

[21] Kiem-Phong Vo. Vcodex: A Platform of Data Tranformers. *Work in progress*, 2004.