Porting Digital's Database Management Products to the Alpha AXP Platform

1 Abstract

The cornerstone software component of high-end production systems is a database management system. Digital has successfully ported the DEC Rdb for OpenVMS relational database management system and the DEC DBMS for OpenVMS network database management system to the Alpha AXP platform. Rdb and DBMS were perhaps the most complex layered products to be ported. The tight coupling of these two products to the OpenVMS VAX system made the port a challenging task. To avoid the future problem of integrating two source code bases, the porting team decided to use a common code base and to overlap current VAX development with the Alpha AXP port. The goal was to provide an easy migration path for software products to the Alpha AXP platform.

Digital is one of a small number of vendors competing in the high-end, complex production systems market. Applications for this market support industries such as banking, stock exchanges, telecommunications, and information services. The Alpha AXP platform is ideally suited to meet the response time, throughput, and availability requirements of these applications, since it offers increased performance while maintaining the superb availability characteristics of VMScluster systems.

Although high-end production systems involve a collection of software packages, the cornerstone software component is a database management system. Digital offers two database management systems for high-end commercial systems: DEC Rdb for OpenVMS, a relational database management system, and DEC DBMS for OpenVMS, a network (CODASYL) database management system. Digital had to port the DEC Rdb for OpenVMS VAX and DEC DBMS for OpenVMS VAX database systems to the Alpha AXP platform as early as possible to continue to compete in this commercial arena. The resulting products are the DEC Rdb for OpenVMS AXP and DEC DBMS for OpenVMS AXP systems. (Since these two products for the Alpha AXP system are the same as those for the VAX system, hereafter, we will refer to the products as Rdb and DBMS.) Additionally, both software products drive many sales of Digital's OpenVMS operating system and transaction processing and information management products such as CDD, ACMS, and DEC RALLY, which integrate with the Rdb and DBMS systems.

Database management systems are among the most complex of all software products. Applications expect these systems to have 7 by 24 availability, sophisticated concurrency capabilities, fast data access, high-speed backup and restore mechanisms, and large buffer pools. To provide such functionality, the Rdb and DBMS products make extensive use of the OpenVMS VAX system, the VAX run-time libraries, and the BLISS and VAX MACRO-32

programming languages. The current release of the product set uses more than 100 system services or run-time library calls. The two products utilize almost every BLISS BUILTIN function, i.e., a machine-specific

function call that generates in-line code. Combined, Rdb and DBMS comprise more than 30 different images. The products run in elevated processing modes, both executive and kernel, and include user-written system services.

Further compounding the complexity of porting the Rdb and DBMS software to the Alpha AXP platform is the fact that they are mature products; DBMS was released in 1981, Rdb in 1984. Because various system capabilities did not exist in the early 1980s, the two database management systems include code that is no longer required. For example, both products have code to move bytes from one data type to another. Also, during image rundown, the products rely on undocumented, operating system behavioral patterns such as the asynchronous system trap (AST) delivery protocols. In addition, the Rdb software contains a modified version of the OpenVMS SORT routine.

Rdb and DBMS were initially designed to run only on the OpenVMS VAX operating system. Consequently, both products heavily utilize VAX-specific features for performance gains.[1] For example, Rdb generates VAX machine code routines as part of query execution plans; the machine code is carefully generated for maximum execution efficiency. This tight coupling of Rdb and DBMS to the OpenVMS VAX system made the port a challenging task.

Since the OpenVMS and BLISS groups were busy with their own porting projects, we in the Database Systems Group had to accomplish our port with little outside help. The task was noteworthy because, by necessity, the team had to port its product set to the Alpha AXP platform earlier than most of the other porting groups. At the same time, Rdb and DBMS were perhaps the most complex layered products that would be ported. Our goal was to port these two products in a timely fashion, so that Digital would truly succeed in providing an easy migration path for software products to the Alpha AXP platform.

In this paper, we first present a brief description of the architecture of the two database management system products. We next describe the guiding policies we formulated to allow the port to proceed as efficiently as possible. Then, we document porting issues that we resolved for the two products. Finally, we summarize our experiences related to this effort.

2 Product Architecture

Digital is unique in the database industry in that we provide two different types of database management systems that layer on top of the same database kernel, which is called KODA. The KODA kernel provides journaling and recovery, locking, access methods (e.g., B-tree, hashing), record and page management, and buffer pool management.

The Rdb software provides language preprocessors, an interactive query front end, a callable interface, catalogue management, query optimization,

and relational operations such as join, select, and project. Rdb supplies a relational interface to the database.

The DBMS product also provides language preprocessors, an interactive query front end, and other software necessary to define, create, and manage data in simple or complex databases. In contrast to Rdb, DBMS provides a CODASYL interface to the database.

Figure 1 shows the relationship of the Rdb and DBMS software products to the KODA database kernel.

3 Porting Policies

Initially, we developed policies to guide our port to the Alpha AXP platform. These policies, which applied to the KODA, Rdb, and DBMS teams, were designed to simplify the port and to ease long-term maintenance requirements.

Common Source Base

Our most important decision was to have a common source code base. That is, we wanted to have one set of source code that could be compiled and run on either a VAX or an Alpha AXP system. At the time we began our port, the OpenVMS group was the only other software group that had started their port, and they had chosen to have two distinct code bases. (The OpenVMS AXP porting schedule dictated the choice.) So with respect to code base, the path we chose was untested. We also decided to maintain common command procedures to compile, build, and link, and common regression tests between the VAX and Alpha AXP systems.

A primary reason for our code base decision was that we did not have the resources to manage two different code bases. Also, although two divergent code sources would have allowed for a stable code base with which to begin the Alpha AXP port, the group strongly wanted to avoid having to merge the two code bases at a future date. Consequently, since our preliminary investigation indicated that a single code base was feasible and that we could hide most of the platform dependencies through the superb macro capability of the BLISS language, we proceeded with the common source code implementation. The single code base allowed us to build and release Alpha AXP and VAX versions of our products at the same time.

Concurrent Releases

Our release schedule complicated the process of adhering to the single code base policy. To meet the schedule, we had to overlap some of the Alpha AXP port with our current VAX releases. That is, the scenario we followed was NOT: work on a VAX release; complete all necessary code changes; stabilize the release; and then create a newer set of sources for the Alpha AXP port. Rather, for the beginning portion of the Alpha AXP port, we also had to change source code destined for a VAX release. Thus, if a module had to be

changed for the earlier VAX release and the same module had already been ported for the Alpha AXP release, the engineer had to propagate the code change to the Alpha AXP source code.

To minimize the effect of double code changes, we first worked on those modules for the Alpha AXP release that were reasonably stable in the current VAX code stream. For example, the BLISS REQUIRE files that we use for data definitions were reasonably stable for the VAX release by the time the Alpha AXP port began. The modules that did not change for the VAX release were also good candidates for helping us to avoid making double code changes. When we finally began to port the bulk of the modules, they were mostly stable and, as a result, only bug fixes for the VAX release required that we manually modify the same module for the Alpha AXP release.

Furthermore, once we began work on the Alpha AXP release, we needed the capability of being able to compile, link, and test on both the Alpha AXP and VAX platforms. So we had to modify our development environment to allow us to identify the code change session as either an Alpha AXP or a VAX session.

No New Functionality

The Alpha AXP release of the database management system product set contains no new functionality. On the first pass, we decided to port the VAX code without designing any new algorithms. We did clean up some code for style, convention, and performance, but basically, the Alpha AXP release remains functionally equivalent to the latest VAX release.

Correct and Fast Code Execution

We did not prioritize our effort to first, be correct, and second, be fast. We decided that we must be correct and fast on certain key issues. For example, on VAX systems, our argument-passing mechanism utilized the argument pointer (AP). To minimize code changes, we could have used the ARGPTR construct in the BLISS cross compiler. However, ARGPTR is inefficient and, therefore, not appropriate for our needs. Consequently, we ensured that our new argument-passing design was efficient, even though doing so was time-consuming.

Minimizing Platform-specific Modules

Code conditionalization, i.e., producing separate code for the VAX and the Alpha AXP platforms, requires various levels of code duplication. For example, the process may require the duplication of an entire module, routines within a module, or certain lines of code within a routine. To minimize the amount of code duplicated, we conditionalized on the smallest code segment possible, using a sensible approach. For example, when forced into using conditional code, we avoided duplicating modules by choosing to keep within a single module. Ideally, we conditionalized just a few lines. Wherever possible, BLISS macros were modified to hide the code conditionalization.

Rdb Is Rdb

We wanted our database management products to "look and feel" the same on an Alpha AXP system as they did on a VAX system. So, to paraphrase from the OpenVMS operating system maxim, we wanted Rdb to be Rdb! That is, the ported Rdb should have the same utilities, the same data structures, the same data definition capabilities, the same data manipulation constructs, etc., as the DEC Rdb for OpenVMS VAX product. Incorporated in this desire for sameness was the fundamental point that we were not going to change the on-disk structures. DBMS was ported with the same goal in mind.

No Changes to On-disk Structures

The KODA kernel stores records on database pages. Unfortunately, the database page is not naturally aligned; page header fields and fields within the records are not aligned. Although aligning these fields would boost performance, to realign all the structures on the database page would require the database to be unloaded and then reloaded. Current customers cannot afford the downtime needed to perform the conversion, so we decided to maintain the same page/record structure. Furthermore, by maintaining the same on-disk structure for the VAX and Alpha AXP databases, we do not preclude future concurrent access to the database in a mixed-architecture VMScluster. Thus, our present design does not require an unload/reload operation, since performing that action would be too much of an impediment to migrating to the Alpha AXP platform. However, we do plan to investigate the potential performance boost from aligned pages/records and, if the gain is substantial, to offer some alignment solution. Note that this section refers only to data structures tied to on-disk structures. We did align all in-memory structures, and we elaborate on this topic in the next section.

4 Porting Details

In this section we describe a general set of issues and solutions that applied to all the groups involved in porting the database management system software to the Alpha AXP platform. We then explain some of the more interesting issues and solutions pertaining to each group.

Common Issues

A collection of general porting issues applied to the Rdb, DBMS, and KODA groups. For example, all groups needed the capability to conditionalize code in a module, so that the compiler on an Alpha AXP system would produce one set of object code, and the compiler on a VAX system would produce another set. Common issues were:

- Varianted code

- Data alignment and field resizing
- Argument-passing mechanism
- BUILTIN functions
- VAX testing

- The CALLG mechanism and AP references
- VAX MACRO-32 modules
- Message file support

Varianted Code. To simplify conditional code, we added a set of literals, for example KOD$K_VAX or KOD$K_ALPHA, that can be used in all our BLISS modules. We could then use these literals to conditionalize code. The code example shown in Figure 2 illustrates the conditionalizing of the PROBE instruction. The PROBE instruction checks the read/write access of a memory location. On Alpha AXP systems, the instruction is quite different from the corresponding instruction on VAX systems. However, BLISS easily handles this difference in a macro, which allows us to change the name and the order of the arguments, pass arguments by value instead of reference, and use an offset instead of a length. By developing such a macro, the actual source code did not have to change.

Data Alignment and Field Resizing. On the first pass, we immediately modified all in-memory data structures so that they were naturally aligned. This step avoided incurring a significant performance penalty on the Alpha AXP platform. In addition, since no single Alpha AXP instructions exist that could be used to easily manipulate bytes or words, many of our in-memory byte (8-bit) and word (16-bit) fields were changed to longwords (32 bits) to reduce the object code size and improve performance.

Once we aligned the in-memory data structures, two groups of data structures remained unaligned: those tied to the database root file, which records database parameters such as associated files and database settings, and the database pages that actually contain the data records. Since the database root file is relatively small (i.e., less than 100 blocks in size), it was aligned also. Thus, the root file is automatically re-created in a conversion that occurs when upgrading a database product to support both the Alpha AXP and VAX architectures. Since this conversion invariably takes place when converting to a newer version of either the Rdb or the DBMS product, the additional realignment of the root is a minor additional expense.

Thus far, we have not pursued any potential modifications of the page data structures, such as aligning them once they are fetched into memory. Note that these structures do not generate unaligned faults. Instead, they force the compiler to generate a few additional instructions to handle the odd alignment.

Argument-passing Mechanism. The VAX and Alpha AXP argument-passing mechanisms are entirely different. Rather than using the standard BLISS mechanism, the existing code depended strongly on the VAX argument-passing

mechanisms by using BLISS macros to reference arguments from the AP. This approach was not possible on Alpha AXP systems due to lack of an AP register. (You could force the AP to be generated, but that process would be slow and would waste memory.) Therefore, we changed our procedure headings to declare a generic formal parameter list (e.g., P1 through PN)

for both the Alpha AXP and the VAX systems and then developed another set of BLISS macros that allowed us to bind to the arguments based on the generated formal parameter list. Since this process involved changing every routine declaration, we developed a text-processing tool that would automatically change the routine headings and thereby avoid the expensive and error-prone task of manually changing each routine.

BUILTIN Functions. Together, the KODA, Rdb, and DBMS code uses most of the BLISS BUILTIN functions. This fact presented a problem for the team porting the software to the Alpha AXP platform. Some VAX BUILTINs were not supported, some behaved differently, and some were eliminated as BUILTINs but emulated by Starlet, an OpenVMS support library. Again, we used BLISS macros to solve the problem. Essentially, our macros categorized the BUILTINs and then performed the appropriate expansion, based on the category. For example, the PROBE BUILTIN differed markedly between the VAX and Alpha AXP implementations, as indicated by Figure 2.

VAX Testing. Another general problem that we had to guard against was the possibility that the Alpha AXP code changes would introduce bugs into the VAX versions of the products. Consequently, we adopted a policy whereby all Alpha AXP changes had to be tested on a VAX system. This policy ensured that we maintained a steady pattern of correct VAX behavior. Also, since the VAX environment was more stable than the Alpha AXP environment, testing on a VAX system helped tremendously in identifying and fixing bugs related to the port.

The CALLG Mechanism and AP References. The Alpha AXP platform does not directly support CALLG, a VAX procedure calling mechanism, and references to the AP. The CALLG mechanism and AP references are slow since they are simulated and automatically allocate stack space to accommodate the largest possible argument list (i.e., 255). In situations where performance was not critical, for example, in an error handler, we replaced CALLG by a standard routine call on both the VAX and the Alpha AXP software versions. When performance was an issue, we used conditional code to retain the CALLG mechanism for the VAX code and to use a standard routine call in the Alpha AXP code. In instances where the CALLG mechanism is used to pass the argument list to the next routine, we constructed an argument vector and replaced CALLG by a special call linkage. The new mechanism passed the pointer to the argument vector by means of a single parameter or a global register. This solution guaranteed good performance on both VAX and Alpha AXP systems yet avoided any conditionalizing of the code.

VAX MACRO-32 Modules. For a variety of reasons, we used VAX MACRO-32 to code some routines in the Rdb, DBMS, and KODA software. For example, basic operations such as record compression, record expansion, and buffer initialization are performed through calls to VAX MACRO-32 routines that are heavily optimized for efficient operation. Some routines are coded in

VAX MACRO-32 for ease of character manipulation. Also, we used VAX MACRO-32 to code machine instructions that were not available through a BLISS BUILTIN function.

We adopted various solutions for these VAX MACRO-32 routines. For those routines where performance was not an issue and BLISS generated acceptable code, we converted to BLISS code. For routines where performance was absolutely critical, we rewrote the routine in Alpha AXP MACRO-64 to utilize the additional registers. Finally, in some cases where we could not rewrite the routine in BLISS code and did not have the resources to convert to MACRO-64 code, we employed the Alpha MACRO cross compiler.

Message File Support. Due to the structure of the database products, as shown in Figure 1, each component has separate message files. Both Rdb and DBMS have a message file that is separate from the KODA message file. Furthermore, the Rdb and DBMS software share the KODA message file.

The message files are merged during the build cycle, so that customers are not required to be aware of the modular layout of the code. As a result, KODA messages, when appended to Rdb's message file, print as Rdb messages (e.g., RDMS-F-msgcode, message text). However, the Rdb source code still references the KODA message codes with the KOD$_ message prefix.

Prior to the introduction of the Alpha AXP architecture, the KODA messages were defined with .LITERAL declarations in the message files. Since we occasionally link images with multiple message files, we wrote a program that would read an .OBJ file and write a new .OBJ file without writing the KODA literal declarations. This process would no longer work since Alpha AXP object files have a different format than VAX object files. As a result, we changed the mechanism to define the KOD$_ symbolic values to be compatible with both the VAX and Alpha AXP architectures.

First, we removed all .LITERAL declarations from the KODA message file. As a result, all KODA messages were defined strictly as RDMS or DBMS messages. Then, after passing the message source file through the message compiler to get the message object file, we invoked the ANALYZE/OBJECT facility to get a listing of the message symbol codes and values for each message. Finally, we wrote a small utility to read the ANALYZE/OBJECT output and generate a BLISS .B32 file, which is shown in Figure 3.

This BLISS program, when compiled and included in an executable image, defines the appropriate KOD$_ message codes and their associated values. This procedure is used on both the OpenVMS VAX and the OpenVMS AXP operating systems to generate the message files. Furthermore, since this group no longer writes programs that read object code, the resulting method is easier to maintain.

The following three sections discuss some problems encountered by each of the porting teams.

Porting the KODA Database Kernel

Among the issues that the KODA group dealt with were those related to calling mechanisms, kernel-mode rundown handlers, and a bugcheck dump mechanism.

Stack-switching/Stall Mechanism. The KODA database kernel performs its own multithreading activities. A single process can be actively attached to multiple databases in the context of a single instantiation of the software. For example, in the DBMS interactive query (DBQ) facility, the user can perform the following operation:

This example has the user attached to two different databases, DB1 and DB2. To issue queries against either database, the user enters the SET STREAM command. In response, KODA establishes the correct data structures and stream context for this database session. This process involves switching data structures and stack context. Consequently, KODA manages its own stack for its executive mode code and data structures. This stack-switching mechanism is complex, and this code is intimately tied to the VAX procedure calling mechanism. For example, whenever a query must stall (e.g., while waiting for a lock request), KODA saves the current executive mode context and then switches back through the stream code out to user mode. This action allows the process to receive user-mode ASTs. This mechanism essentially saves a call frame so that after the user-mode stall has completed, KODA can set up the appropriate stack and return to the calling routine by means of the saved call frame.

The calling/return mechanism is entirely different for the VAX and Alpha AXP architectures. On Alpha AXP systems, for each routine, the compiler generates prologue code and epilogue code to manage the routine calling mechanism. Accordingly, the KODA stack mechanism had to rely on this new mechanism. In addition, for this level of support, the routine that was coded in BLISS for the VAX platform had to be coded in MACRO-64 on the Alpha AXP platform.

Kernel-mode Rundown Handlers. Another example of KODA's close tie to OpenVMS behavior involved the use of KODA's kernel-mode rundown handler. On VAX systems, in the event of an abnormal failure, we must clean up certain data structures and release resources such as locks or channels. Furthermore, database recovery must start before the image rundown is completed, so that surviving processes cannot acquire locks on resources before the databases are recovered.

We accomplish this image cleanup through the use of a user-defined system service (i.e., a system service not defined by the OpenVMS system), which acts as a kernel-mode rundown handler. In addition to releasing database resources, the handler also cleaned up OpenVMS data structures such as

the pending AST queue. These OpenVMS data structures changed significantly for the Alpha AXP architecture. For example, an Alpha AXP system has five pending AST queues instead of one. In addition, this handler routine would acquire the OpenVMS scheduler spinlock and perform "poor man's lockdown," which effectively pages the entire routine into memory (since the code

cannot incur a page fault at elevated interrupt priority level, IPL). For Alpha AXP, code and data cannot be located in the same PSECT, so this trick was not possible. Instead, we used the $LKWSET macro to lock pages in memory and then to clean up the OpenVMS data structures.

After we completed and tested the code, the database and OpenVMS engineering teams decided that such intricacy was needlessly complex, and that the OpenVMS AXP software could clean up the data structures based on its image control block and related structures. This example shows how the OpenVMS AXP system offers different functionality than the OpenVMS VAX system, i.e., the port offered the opportunity to clean up existing mechanisms.

Bugcheck Dump Mechanism. Complex, sophisticated software products are by nature difficult to debug. Most of these products utilize a data structure dumping mechanism whenever an internal software or hardware error is encountered. KODA has a mechanism called a bugcheck dump that performs this service. When an unexpected exception is generated, the bugcheck dump code prints all relevant data structures into a file. In addition, the dump includes a stack dump. On VAX systems, the bugcheck dump traces back down the stack using the saved call frames and prints out all the fields in each call frame, the routine name, and the arguments passed.

In particular, the method for printing the symbolic name of the routines is especially clever. After linking an image, we utilize a program that scans the symbol table (.STB file) produced by the linker. Then the program creates its own object file, which includes a relative offset of all the routines and their symbolic names. Finally, the image is relinked, and this new object file is included into the image in a particular PSECT. When tracing back down the call frames, the bugcheck dump also checks the special PSECT to locate and print the correct routine name. This dump is an invaluable tool in determining the causes of unexpected errors. Figure 4 includes two routine calls from a stack trace, indicated by the lines of code that begin with "Saved PC."

Alpha AXP systems have no equivalent to the VAX call frames, so it is impossible to use the call frame mechanism to trace down through the stack. As mentioned previously, Alpha AXP routines utilize prologue and epilogue code for returning from routine calls. Procedure descriptors contain information such as entry address and register save information.

On Alpha AXP systems, another Digital group supplied a set of routines that allows tracing the call sequence. This set provided the basic capability to print the routine calling sequence that led to an abnormal exception. In addition, the Alpha AXP linker produced a symbol table file. However, we decided to simplify our bugcheck mechanism. Although we still search the symbol table file for all routine addresses, rather than create an Alpha

AXP object file, we create a VAX MACRO-32 file that includes the routine name and address/offset. Then, we simply use the Alpha MACRO cross compiler to generate the Alpha AXP object, which gets linked into the image on the second pass. In fact, we changed our VAX bugcheck routine to produce a

MACRO-32 file with routine name and offsets. This process is simpler than directly creating an object file, as we did previously.

Even though the routines provided this call trace-back capability, we were missing the arguments passed to the routines, perhaps the most important part of the stack trace. The VAX mechanism captured this data, because very often a bugcheck results from one routine passing an improper argument to another routine. The Alpha AXP system does not provide a way to capture this information, because the routine calling sequence reuses registers R16 through R21 for passing arguments.

Porting Rdb

Some issues handled by the Rdb porting group were associated with the dispatch code, Alpha AXP code generation, Rdb precompilers, and Rdb system relations.

Dispatch Code. The dispatch code is the topmost layer of the Rdb software and is called directly by the user application by means of relational call interface (RCI) calls.[2] The main function of dispatch code is to direct the user request to the correct target Rdb executive (local or remote) for processing. On VAX systems, the dispatch code passes the user arguments to the Rdb software using the CALLG linkage.[3] On Alpha AXP systems, CALLG linkage is very inefficient. Therefore, the dispatch code was changed to build a user argument vector in the same style as the VAX argument list, and the pointer to the argument vector was passed as a single parameter. The code in Rdb was changed to bind to the user arguments using the offset from the pointer to the argument vector.

Using two different calling mechanisms in the dispatch to pass user arguments was a careful design. On VAX systems, the existing CALLG mechanism was retained to ensure backward compatibility between different versions of the Rdb dispatch, Rdb layered products, and gateways. A new calling mechanism was used on Alpha AXP systems to ensure good performance, since every user request to the Rdb executive goes through the dispatch.

Code Generator. Rdb uses compiled BLISS code and generated machine code to execute user requests. During request compilation, Rdb generates highly efficient routines using the target machine instructions. These routines perform basic data operations including data conversion, data movement between buffers, aggregation, and expression evaluation.

The design of the Rdb code generator to produce Alpha AXP machine code was undoubtedly the most complex porting task. Use of a mechanism other than code generation would have reduced the porting effort. However, at the time we began porting Rdb, it was not clear if an alternate mechanism would guarantee an acceptable level of performance. Good performance was

considered critical to the success of Rdb on Alpha AXP systems. Therefore, we decided to add functionality to the Rdb code generator to produce Alpha AXP code. To generate efficient Alpha AXP code sequences, we observed specific guidelines.[4]

On Alpha AXP systems, code that references data items with increasing memory addresses executes more efficiently. Therefore, the algorithm was changed to first order the data items by increasing memory addresses and then generate code to process the data.

In Rdb, each data item has a null bit that indicates whether or not the value of the data item is known. As shown in Figure 5, to conserve space, the null bits of different data items are stored together like a bit vector within a record. Loading/storing a null bit is an expensive operation on Alpha AXP systems.[4] Therefore, the algorithm was modified to fetch a batch of null bits into a register. When all null bits in the register are processed, the batch is written and the next batch of null bits is fetched. This approach reduced the number of load and store instructions and made the code sequence much more efficient.

On Alpha AXP systems, the machine code routines generated by Rdb use four different addressing modes to access data items: absolute address, base register plus offset, integer register content, and floating-point register content. Each of the Alpha AXP registers R12 through R15 is used as a base register. Thus, any data stored within 256K (4 x 64K) of memory space can be accessed efficiently. To maximize data access efficiency and caching, changes were made in the code generator to allocate data densely. To improve performance further, data items were allocated at quadword or longword aligned addresses.

An Alpha AXP code sequence executes more efficiently when instructions can be multi-issued and executed in parallel. This can be achieved by reordering the sequence of instructions while maintaining any chronological dependency between instructions. To take advantage of this Alpha AXP feature, BLISS macros were developed to reorder and interleave the instructions in a generated code sequence.

On Alpha AXP systems, backward branches in the code slow down the execution because of instruction stream invalidation.[4] Changes were made in the Rdb code generator to minimize backward branches. This change at times increased the size of the generated code but improved the code execution efficiency. Further, Boolean code generation algorithms were modified to incorporate branch prediction logic; code sequences with a smaller probability of execution were branched out of the main code stream. This technique maximized the effect of instruction stream caching.

Rdb Precompilers. An Rdb precompiler preprocesses a user application program that includes Rdb statements and replaces these statements by standard RCI calls to the Rdb software.[2] The Rdb statements embedded in the applications can be one of three types: structured query language (SQL), Rdb preprocessors language (RdbPRE), or relational data manipulation language (RDML). There are three different Rdb precompilers to support

these languages.

The SQL precompiler, an industry-standard language interface to Rdb, is a strategic Rdb component. A long-term goal of this precompiler is flexibility in future developments and ease of maintenance. To meet this goal, the SQL precompiler was redesigned to use the GEM compiler on Alpha AXP systems to preprocess SQL application programs and produce Alpha AXP object code.

The RdbPRE precompiler is a proprietary language interface to Rdb. The long-term goal is no new functionality and minimal maintenance. So the main objective was to reduce the effort required to port this compiler. This was achieved by retaining the existing design and using the Alpha MACRO cross compiler to produce Alpha AXP objects from VAX MACRO-32 files.

The RDML precompiler is also a proprietary language interface to Rdb. Unlike the RdbPRE precompiler, this compiler does not produce VAX MACRO-32 files. So porting it was an easy and straightforward task.

Rdb System Relations. Rdb uses system relations to record information about the user relations and the database. The system relations are stored on disk and loaded into memory on demand. Since they are frequently referenced during user request processing, efficient access to data in system relations is critical for performance. On Alpha AXP systems, accessing data from memory is efficient if it is located on either a longword or a quadword address boundary.[4] Therefore, changes were made to the in-memory system data structures to align each data field to at least a longword address boundary. Further, data fields that were a byte or a word were expanded to a longword.

The data in system relations was accessed by using RdbPRE statements embedded in Rdb source modules. Porting such Rdb modules posed a dilemma. To compile these modules, first the RdbPRE compiler had to be ported to the Alpha AXP platform. Vice versa, to port and test the RdbPRE precompiler, Rdb had to be ported and running on the Alpha AXP platform. Moreover, RdbPRE was no longer a strategic language interface. Therefore, new BLISS macros were designed that replaced the embedded RdbPRE statements.

Porting DBMS

This section discusses some experiences of the DBMS porting group, namely those related to the Database Control System (DBCS) interface, the H_FLOAT data type support, and the use of the Alpha User-mode Debugging Environment (AUD).

DBM$32, the Primary Interface to the DBMS. The DBCS for the DBMS software uses a single subroutine (DBM$32) as its primary entry point. This entry point is used by the DBMS precompilers (FDML, for FORTRAN, and DML, for other languages except COBOL), as well as other layered products, such as

COBOL and DATATRIEVE.

After receiving control, DBM$32 performs some processing and then, using the CALLG mechanism, passes the entire argument list to lower-level routines for further processing. These lower-level routines, in turn, often pass on the argument list, sometimes as deep as five or six levels.

Because we found CALLG to be inefficient, we decided to change the primary entry point into the DBCS. Rather than passing up to 26 separate arguments, DBMS creates a vector of longwords; each longword contains an argument that would have been passed using a parameter. Once this vector is created (often during the compilation phase for the precompilers), DBM$32_VEC (the VECTOR version of DBM$32) is called with a single parameter: the address of the argument list. An example is shown in Figure 6.

Layered products using DBMS were advised of the new interface and were requested to use it as soon as possible. However, since the changed interface was incompatible with some existing products, the old interface was retained. DBM$32_VEC uses the new interface, and DBM$32 homes the argument list (thus creating the above vector) and then passes that, by reference, to DBM$32_VEC.

Support of H_FLOAT Data Types. The H_FLOAT data type is fully supported on the VAX processor, but the Alpha AXP processor has no high-precision floating-point formats. Although facilities exist on Alpha AXP processors to read an H_FLOAT data type, no such facility exists to write an H_FLOAT data type.

As a result, DBMS customers are advised to eliminate any H_FLOAT data in databases before moving them to an Alpha AXP system. The DBMS Database Restructure Utility (DRU) can be used to change all H_FLOAT data to another common floating-point format.

In preparation for mixed VAX and Alpha AXP VMScluster systems, DBMS was modified such that databases with H_FLOAT data can still be accessed. However, a run-time conversion error occurs if H_FLOAT data is accessed from an Alpha AXP system.

Use of AUD. The Alpha User-mode Debugging Environment is a set of facilities that aids testing and debugging of native Alpha AXP code on any OpenVMS VAX system. AUD allowed as much Alpha AXP user-mode code as possible to be ported immediately to the Alpha AXP system and to be substantially debugged before Alpha AXP hardware was available. Early in the DBMS porting effort, we used AUD to verify our port and to ensure that our code was working correctly.

However, several issues hampered the success of using AUD in porting the DBMS software:

1. DBMS makes frequent use of signaled exceptions. AUD had difficulty in handling exceptions that cross the boundary between the Alpha AXP and VAX systems.

2. DBMS uses special stack manipulation code (stream code) to perform multithreading functions. AUD would become confused if the stack were to change unexpectedly.

3. At the time we were using AUD, the DBCS had been ported, but KODA (i.e., the low-level services used by the DBCS) had not. As a result, many variables needed to be defined as crossing the boundary between the Alpha AXP and VAX systems. The setup time to define this information was significant.

4. Since the code was still running on a VAX processor, many VAX dependencies were not caught by AUD. In particular, system services that changed in subtle ways would work as before because the operating system was still the OpenVMS system.

5. Most of the changes that we made in DBMS were not conditional, that is, the changes would affect both VAX and Alpha AXP systems. As a result, we were able to test our code on VAX systems with a fairly high degree of certainty that our code was correct, barring any operating system or compiler bugs.

We did eventually get an AUD version of DBMS working. However, since we spent a considerable amount of time accomplishing this, and we did not actually find any bugs in our code by using AUD, we decided not to use AUD in further areas of DBMS.

Shortly after using AUD, we received our Alpha Demonstration Unit (ADU) and could test our code on actual Alpha AXP hardware. The only problems we found, which were missed during our initial port, were VAX-style argument list assumptions. Some of our code assumed that routine arguments were contiguous in virtual memory; on Alpha AXP systems, this is not the case.

5 Conclusion

To conclude the paper, we discuss our plans for performance testing and our reflections on the porting process.

Performance

We have only begun our performance tests. Currently, we are running the TPC-B performance benchmark. We also plan to test against all TPC benchmarks (A, B, and C) and other benchmarks such as the Wisconsin benchmark. We are trying to minimize the amount of time spent in PALcode, decreasing the code path length, reducing the cycles per instruction, and optimizing internal algorithms.

Planned testing will also evaluate the effect of additional data alignment.

As mentioned earlier, the ease-of-migration issue is paramount for our current customers. Consequently, we have not realigned the database pages because that action would require too much downtime. Nevertheless, we do not want to preclude new customers, or current customers who need the performance boost, from utilizing a properly aligned database page. To test

the potential performance improvement, we plan to create a test database that is completely aligned, in memory and on disk, and compare the TPC performance against the standard database.

Reflections

At the beginning of the paper, we stated that our goal was for Digital to provide an easy migration path to the Alpha AXP platform for software products. Although we encountered some difficulties, we believe our Rdb and DBMS porting efforts attest to Digital's success in this endeavor.

As one example of how the experience influenced our approach to porting, we had to learn new methodologies, practices, and system behavior on the Alpha AXP machines. For instance, when stepping through a particular code sequence with the debugger, we would end up in an infinite loop; if we just ran the code, the sequence would work. Although this behavior was documented, we encountered the problem several times before we fully understood the ramifications and appropriately changed our development methods.

Overall, the porting effort had the following positive results:

- The port allowed us to clean up our code, even though we tried to avoid algorithm changes. Because we had to port and review every line of code, we managed to move the code to a more consistent coding convention.

- The port acted as a learning experience for most of the engineers. Most mature products contain some code that has not been modified in years. The port forced us to review and understand such code sequences. As a result, we ended up with more knowledgeable engineers.

- The port allowed us to transform the code into a more portable state. As we moved away from tight ties to VAX behavior, we simplified future tasks such as moving to the OSF/1 and Windows NT operating systems.

- Although overlapping current VAX development with the Alpha AXP port slowed down the porting process, the decision to use a common code base eliminated the future need to integrate two divergent source codes.

- Surprisingly, the code did not grow appreciably in size or complexity. One strength of the Rdb and DBMS software has been the ability to easily modify the code and to add new functionality. Even after the port, we find that the products are as malleable and as easy to modify as before.

- We found unreported bugs in our VAX products.

Virtually all the groups involved did a masterful job. The program team and

various Alpha AXP committees anticipated potential issues and ensured that the program proceeded smoothly and predictably. The cross compilers from the language groups worked superbly. The OpenVMS AXP and hardware groups delivered their products on time, and when a user logs in to an Alpha AXP system, the OpenVMS AXP system is not only familiar but faster.

6 Acknowledgments

The successful port of the Rdb and DBMS software to the OpenVMS AXP operating system was a result of the contributions made by many of the engineers in the Database Systems Group. The authors sincerely acknowledge the effort of each engineer in achieving the project goal, that is, to be able to quickly offer correct versions of Rdb and DBMS on the Alpha AXP platform. Finally, an unsung hero in the company-wide effort was Digital's VAX Notes communications facility. VAX Notes functioned as an excellent medium for identifying and sharing problems and solutions.

7 References

1. T. Leonard, VAX Architecture Reference Manual (Bedford, MA: Digital Press, Order No. EY-3459E-DP, 1987).

2. DSRI Handbook (Maynard, MA: Digital Equipment Corporation, Order No. AA-GV71A-TE, 1986).

3. OpenVMS Calling Standard (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQY2A-TK, 1992).

4. R. Sites, ed., Alpha Architecture Reference Manual (Burlington, MA: Digital Press, Order No. EY-L520E-DP, 1992).

8 Trademarks

9 Biographies

Jeffrey A. Coffler A principal software engineer in the Database Systems Engineering Group, Jeff Coffler led the effort to port DBMS to the Alpha AXP platform. Prior to this, Jeff worked on the DBMS and Rdb backup/restore facility and on new DBMS features and maintenance. He is currently working on the project to port Rdb for OpenVMS to operating systems such as Windows NT and OSF/1. He has also contributed to the RSTS/E operating system,

WPS-PLUS porting, and workflow management projects. Jeff joined Digital in 1984 and holds a B.S.C.S. (1983) from California State University at Northridge.

Porting Digital's Database Management Products to the Alpha AXP Platform

Zia Mohamed Zia Mohamed has been a member of the Database Systems Group since joining Digital in 1989. He works in the area of query optimization for the DEC Rdb for OpenVMS products; his contributions involve cost-based optimization of database queries and algorithms for execution of optimized query plans. He has developed dynamic OR optimization techniques, refinement of cost-model, and algorithms for better access plans for views. Zia holds a B.S. degree in electrical engineering from Bangalore University, India, and an M.S. degree in computer science from Texas Tech University.

Peter M. Spiro Peter Spiro, a consulting software engineer, is currently the technical director for the Rdb and DBMS software products. Peter's current focus is database performance for Alpha AXP systems and very large database issues. Peter joined Digital in 1985, after receiving M.S. degrees in forest science and computer science from the University of Wisconsin-Madison. He has four patents related to database journaling and recovery, and he has authored two papers for earlier issues of the Digital Technical Journal.