

OpenVMS Technical Journal V6

Porting OpenVMS to HP Integrity Servers



Porting OpenVMS to HP Integrity Servers	2
Overview	2
The Early Days	2
Research	3
Big Decisions	4
Technology Overview	6
The Plan	7
Development Begins	9
Boot Contest	14
OpenVMS V8.0 – June 30, 2003	16
OpenVMS V8.1 – December 18, 2003	17
OpenVMS V8.2	18
Summary	20
Acknowledgements	20
For more information	21

Porting OpenVMS to HP Integrity Servers

Clair Grant, OpenVMS Base Operating System Technical Leader

Distinguished Technologist, Hewlett-Packard Company

Overview

This article describes the 3.5 years of work that went into porting OpenVMS to Integrity Servers. A little history, the rationale behind the major decisions, and some technical details are combined to present the engineering that resulted in OpenVMS I64 V8.2. First there is a chronology of the major events and decisions of the project. The concluding sections provide details of some of the technology that we found the most interesting and challenging and how they came together to produce the preliminary and final releases.

The Early Days

Decision

During the first half of 2001, a group of senior COMPAQ engineers completed an evaluation of the Alpha processor. Alpha had held a performance advantage over Intel processors for several years. However, the difference was getting smaller as time progressed. The evaluation team concluded that even if all planned Alpha projects completed on schedule the two processor families would be performance equals in approximately 2005 and Alpha would almost certainly fall behind in the future.

The economics of future Alpha development were simply no longer feasible. The work on the EV7 that was in progress would continue, but that would be the end of Alpha processor development and the associated hardware platforms would be the end of the Alpha line. All EV8 processor development was terminated.

Announcement

On Monday June 25, 2001, senior COMPAQ management publicly announced that Alpha development would be terminated and that OpenVMS would be ported to the Intel Itanium architecture. The message to customers was clear: in the future there will be faster systems at a lower price than if the current Alpha plans were continued. They also said when it would happen:

“We will deliver a production quality release of OpenVMS in 2004.”

This was a shocking announcement, not only for the public but also for OpenVMS Development!

The promise to application providers was that they would “Recompile, Relink, and Go.” A popular variation of this became, “You are a qual away” from running on HP Integrity. This was an extremely bold message, since we had not even begun to research the details of what needed to be done. However, it became an extremely powerful measuring stick for implementation decisions made throughout the project.

Another aspect of this huge undertaking was that the existing Alpha OpenVMS roadmap would remain unchanged. The importance of this should not be underestimated. It was critical to our customers. Plus, a large number of engineering resources (engineers, build team, release team, business management, qualification and verification, documentation) were already committed to the next OpenVMS release and that could not change.

NOTE: Within just a few days of the announcement I received many mail messages saying, in one form or another, “You people are crazy! VMS will never run on a processor with only two privilege modes!” I attempted to reassure them that “Itanium is indeed a four-mode processor, just like VAX, and VMS will be just fine.”

Research

June 25, 2001 was not only the day of ‘the announcement,’ it was also the day we started working on the project. First, it was communicated to our customers and application providers what we were about to do and what it would mean to them. We also met with engineering and field organizations that would be affected by this new direction. Finally, we formed the initial team of engineers to do the research that would be the input for the plans and schedules.

Read the Manuals

The Intel websites have a great deal of relevant information: white papers, current hardware offerings, future plans, etc. We started reading the [Intel IA-64 Architecture Software Developer’s Manual](#), a 4-volume set that would be, and still is, our primary source of information for the architecture (available from <http://www.csee.umbc.edu/help/architecture/>). The Advanced Configuration and Power Interface (ACPI) specification from Intel was also critical. But we knew this would not be sufficient; we needed to talk with those who architected and implemented the system.

Question: When is a word not a word?

Answer: When you put the Intel Itanium terminology and Alpha terminology side by side.

A byte is 8 bits in both architectures, but that is where the commonality in data sizes ends. The VAX/Alpha “word” is 16 bits long, whereas the Intel “word” is 32 bits. (See Appendix A - How to Make Your Head Hurt for the complete comparison.) Once we got beyond the “communication gap,” we wondered how we would document such a fundamental difference. Finally we decided that we would continue with our long-standing tradition of documenting only OpenVMS, and ignore the terminology difference completely. It was the easy way out, but every alternative we considered led to an ugly documentation strategy that did not make anything clearer.

Talk with Engineers

Some members of the COMPAQ compiler group had Intel experience as a result of previous projects. The GEM backend used by the OpenVMS compilers was already “Itanium-savvy.” We sought the advice of the compiler developers as to how we should get started. They would soon play a critical role, because compiling the code would be our first step in the porting process.

Just as we were starting to gain a little “knowledge momentum” and realizing where some major decisions might be needed, the next “announcement” occurred: COMPAQ and HP were proposing to merge. This was announced just days before the annual COMPAQ Enterprise Technology Symposium, so the entire tenor of that event changed over night. Little did we know at the time that it would take eight months for the merger would be completed. It was not until May 7, 2002 that OpenVMS become part of Hewlett-Packard.

Historical Footnote: The first public engineering presentation of OpenVMS on the Intel Itanium Processor Family was at the annual COMPAQ Enterprise Technology Symposium in Anaheim, CA. The presentation was scheduled for Tuesday morning – September 11, 2001. However, the world stopped that morning and watched in horror at the tragic events at New York City’s World Trade Center, western Pennsylvania, and the Pentagon in Washington D.C. The entire COMPAQ conference resorted to a make-shift schedule for the rest of the week.

Part of the “First Announcement” was that Tru64 UNIX would also be ported. Some of the Tru64 engineers had Intel experience from other projects. Since we were all in this together, we jointly invited a small group of Intel architects to Nashua in mid-September of 2001 for a 3-day seminar covering the instruction set, operating system interfaces, the boot path, and ACPI. It was a real eye-opener! The manuals had brought us a long way with the basics but the face-to-face conversations made it clear that we still had a long way to go. Fortunately, we now had engineers to talk to whenever we had questions.

HP had codeveloped the Itanium architecture with Intel so all of HP’s systems were moving in that direction and we were already porting OpenVMS to the desired platform. Immediately, on Tuesday

May 7th, we were called by HP firmware engineers offering to help us in any way, and we also started conversations with the HP-UX engineers. Another May 7th event was the joining of the COMPAQ and HP networks and e-mail. Access to documents and engineers was just a few key strokes away. It was gratifying, as well as tremendously satisfying to realize that in a matter of hours we had access to large development organizations who were eager to share their knowledge with us. After all, they had already overcome many of the hurdles we were about to encounter. Their knowledge would prove invaluable.

The book [ia-64 linux kernel](http://www.lia64.org/book/) by David Mosberger and Stephane Eranian was extremely helpful. In the spring of 2002, OpenVMS engineers attended a seminar in Marlboro, MA, presented by Mr. Mosberger. An excellent reference for anyone wanting to learn about the Intel architecture, the book is available from <http://www.lia64.org/book/>.

Big Decisions

Very early in the project, we had to make some big decisions that had significant technical implications for porting the code. In some cases, these decisions would be visible to our customers. The following sections describe many of those decisions, but always there was the overriding theme that we stated at the beginning: this was not going to be a "bug-for-bug" compatible port of the existing Alpha code. We would make decisions, even drastic ones, if they would make a better OpenVMS in the future. As you will see, we pushed the limits a few times, but the quality of the product and the needs of our customers was our paramount concern.

1. Implement the Intel Calling Standard (and More)

This was a completely unanticipated outcome. After lengthy evaluation we stunned everyone, including ourselves, when we decided to implement a variant of the Intel Calling Standard. In addition, we decided to use the more universally known Executable and Linking Format (ELF) and Debugging with Attributed Record Formats (DWARF) standards rather than our unique OpenVMS standards.

The calling standard decision was based on a combination of considerations. For the benefits of performance and shared software technology, we tried to use the Intel conventions as much as possible, straying from them only where necessary in order to preserve user-visible characteristics that are essential for multilanguage compatibility and interoperability, features that came from the original VAX design. We knew that creating such a hybrid design was a monumental decision, but over the next couple of years we came to realize that even "monumental" was a gross understatement!

The decision to use the Intel calling standard required us to create the Calling Standard Committee, which consisted of compiler, run-time, and OS experts. Their job was to systematically evaluate every detail of the OpenVMS Calling Standard for VAX and Alpha systems and work out what should be preserved, what had to change, and how best to support the "Recompile, Relink and Go" message. This decision had to be communicated early to Intel compiler developers because the OpenVMS C++ and FORTRAN compilers would be coming from Intel.

2. ELF/DWARF File Format

Selecting the ELF/DWARF file format caused enormous shock waves. The compilers, the linker, the librarian, the debugger, the dump analyzer, and the image activator (just to name a few) would have completely different formats of input and/or output. But we wanted to make OpenVMS more receptive to the world of porting applications and analysis tools. The system would be more accessible to applications because it would have more well-known internal formats and more developers would understand them.

One of the goals of the project was to make OpenVMS more portable. Who knows, maybe someone will be doing this again! The calling standard, ELF, and DWARF decisions doubled the length of the port of OpenVMS, not just in the coding time but also in the conceptualization and the debugging of completely new implementations. It was hard work, but OpenVMS is a better system for it.

3. No Compiler for Alpha Assembler Code

Half of OpenVMS is written in MACRO-32, so we needed a compiler. On Alpha we have the AMACRO compiler, so we created the IMACRO compiler on Integrity. As for Alpha assembler code, we concluded that there isn't much of that around, especially in the application world, and there probably would not be any more written. We could make it easy to rewrite most Alpha assembler by providing the right compiler built-ins for instructions and the right library routines for internal calling standard knowledge, like stack unwinding. These seemed to be the two reasons most implementers wrote Alpha assembler code.

The Integrity C compiler would not be supporting the asm construct. We validated our theory by rewriting all of our existing asm code in C; that is, we never replaced a C asm that contained Alpha assembler with Intel assembler code. This was a very good sign and, although it was not a silver bullet for all applications, it confirmed our "no compiler" decision.

4. Binary Translator for Alpha Images

On Announcement Day we had said that there would be no binary translator, as there is for porting from VAX to Alpha. The binary translator is a very large and difficult piece of code. This decision had implications for numerous components of the system. Components of the base operating system, some RTLs, and even the Alpha firmware have specific code just for the binary translator. Not creating a translator would have saved us a lot of work, but it became clear that it was necessary for the customer offering. We contracted this work to Software Resources International (SRI), since they specialize in this type of application and already worked with us on other projects.

5. Ada – Yes; PL/I –No

OpenVMS is written almost entirely in MACRO-32, BLISS, and C, and we knew that a little Intel assembler would be necessary. We could go a long way in the port with nothing else. However, we do have a little Ada code, and it is popular in a few OpenVMS customer segments. We decided to contract with Ada Core Technologies to provide an OpenVMS I64 Ada compiler.

But we decided against a PL/I compiler because there is very little PL/I in the OpenVMS application world and PL/I-to-C translators are available. (In the end, we did rewrite the PL/I part of MONITOR in C! The result is common code for Alpha and Integrity.)

6. Default Floating Point Format is IEEE

The Floating Point format was a gigantic decision that came with far more ramifications than we originally anticipated. On Alpha, VAX floating point formats (commonly known as F/D/G) and IEEE are all implemented in the hardware. In other words, there is no performance advantage to convert to IEEE. For OpenVMS on Integrity, the VAX floating point formats are emulated in software —only IEEE is in the hardware. Depending on the application, the performance difference can be significant.

We decided to make IEEE the OpenVMS default on Integrity. If you compile without a format qualifier on Alpha, you get VAX floating points. On Integrity, the same compile command will produce IEEE floating points. Although this affects some calculations in the highest degrees of precision, we could not justify the performance hit from defaulting to emulation. For those applications where the slower software emulation is an acceptable solution, the OpenVMS Integrity compilers support F/D/G by using the /FLOAT switch.

7. Math Library

The math runtime library is one of the most visible performance-critical components of the system. On Alpha, it is written in C specifically for the Alpha architecture. Recompiling it on Integrity was not acceptable. A viable alternative appeared to be the Intel Itanium math library, which is optimized for the Integrity architecture. We decided to get the math library from Intel. Part of the announcement included sharing compiler technology with Intel, and this was one of the first cases where we reaped

the benefits of it. The drawbacks were minor, and vastly overshadowed by the performance improvements, and we were better prepared for future changes.

8. Expanding the Kernel Process Model

On Alpha, a number of OpenVMS system components, such as RMS, the XQP, and DECnet, managed multiple threads of execution on their own; that is, they each knew the details of a context swap and implemented the pieces their individual environments needed in order to suspend and resume execution contexts. Due to different "context" details, these implementations would have to be rewritten for Integrity. We knew some customers had also made similar design decisions. These changes are difficult on any system, but they are particularly challenging on Integrity. Considering the number of components we would need to modify, this would be a daunting task. Likewise, it would be a huge task for customers porting to Integrity. Therefore, we decided to enhance an internal mechanism called "kernel processes" to be callable from any mode and modify all of the OpenVMS components with private threading to call the enhanced routines. Writing the extremely complicated code once and then converting the callers was a much better approach and it would increase the maintainability of the code immensely. An additional benefit would be common code for Alpha and Integrity. We have a paper with examples (google EXE\$KP_START) about converting to the new routines (highly recommended) rather than porting their Alpha code.

9. Licensing

We were concerned about being a new HP offering. Licensing OpenVMS was totally different from any of the company's existing products. This would not be necessary if we were willing to make a rather large change. For Integrity (but not Alpha) we decided to adopt the licensing model of HP-UX, in which you purchase a level of operating environment (OE). This would require significant work on our part but it was the right strategy to pursue.

Technology Overview

The vast majority of code in a large and complex operating system like OpenVMS has no dependency on the platform or CPU architectures. For example, device drivers, network protocols, clusters, lock manager, and the file system required no recoding to run on Integrity. However, we had some very big architectural challenges and there was a great deal of work to do in a few concentrated areas of the system.

The next few subsections describe the components of the base operating system that made up the bulk of our work. The components fall into one of the following categories:

1. Those that are analogous to Alpha but different in detail.
2. Those that are unique to Integrity.

No Alpha Console

On Alpha, the console is part of the firmware package. It plays an integral role in booting the system and in device discovery for OpenVMS. The Intel Itanium architecture boot path is by necessity a generic and minimal environment, since its goal is to accommodate the basic needs of a number of operating systems without being specific to any. Therefore, the boot path would be completely new code up to the point of starting SYSBOOT. On Integrity, OpenVMS itself provides many of the runtime services that are provided by the firmware on Alpha.

No Alpha Privileged Architecture Library (PAL)

The PAL is another part of the Alpha firmware package. While the Alpha architecture is operating system agnostic, it allows an operating system to provide its own specialized PAL. The OpenVMS PAL contains code for the VAX queue instructions, the VAX special registers, VAX IPL and privilege mode change management, interrupt handling, knowledge of PTE format for translation buffer misses, and alignment faults. Thus, code written specifically to use these features on a VAX can run unchanged on an Alpha. Again, OpenVMS itself would need to provide these functions on Integrity. Replacing the Alpha PAL was especially challenging.

Porting OpenVMS to HP Integrity Servers - Clair Grant

Different Processor Primitives

The Itanium register set is very different from Alpha, and vastly bigger. The Register Stack Engine is a mechanism to help the operating system manage this large stack environment. It required totally new code in OpenVMS.

The load-locked/store-conditional instruction pairs on Alpha guarantee atomic access to data in a multiprocessor environment. Like VAX, Itanium has individual instructions that guarantee atomicity.

The new register set changes the way a process's context is saved. Therefore, any code that did its own "threading" would need serious revision.

The Alpha PAL does a lot of interrupt processing before notifying the operating system of the interrupt. One of the biggest parts of the porting project was the Software Interrupt Services (SWIS), Integrity-specific code that completely controls interrupt processing and its associated mode checking and management.

Calling Standard Impact

The Intel Calling Standard's register conventions and stack usage are very different from those of the Alpha Calling Standard. To avoid modifying many source code modules due to the register differences, especially modules written in MACRO-32 and BLISS, we defined a mapping of Alpha's R0-R31 to Itanium's R0-R31. The compilers automatically make the conversion, based on a switch. However, programmers still need to know the register mappings when debugging. This mapping found its way to scraps of paper taped to our workstations, from there to a PowerPoint slide, and finally onto a mouse pad, which is still an invaluable reference. (See Appendix B - OpenVMS Alpha to Itanium Register Mapping for the entire register set.)

Integrity exception processing uses "PC mapping" rather than the "frame pointer" scheme we are familiar with on VAX and Alpha. Furthermore, there are two stacks to be coordinated: an integer register stack managed by the hardware as well as the memory stack managed by the generated code. On VAX and Alpha, programmers "walk the stack" in order to find information. The Integrity memory stack has very little information of this type. To find a caller's saved registers, programmers use the current PC to index into the "unwind tables," which point to the saved registers. The unwind data created by the compilers is loaded into the unwind tables during image activation. Use of the unwind tables is the foundation of exception handling.

NOTE: We made the LIBRTL calling standard routines usable from all privilege modes so there would be a single OpenVMS source for this knowledge.

It seemed to take forever to get the base operating system and the LIBRTL unwind interfaces correct. For months we received a steady stream of exception handling problem reports. Even after the code functioned correctly, it was reworked a couple of times to improve its performance. As we expected, this was a very difficult part of the project.

ELF/DWARF Impact

The decision to use the ELF/DWARF internal format resulted in a lot of new code and also made our progress difficult to plan because the areas most impacted were our development tools. Creative debugging ideas were at a premium, to say the least.

The Plan

During the latter months of 2001 we assembled the pieces of the development plan and plotted a preliminary schedule. This was necessary in order to understand the early dependencies. It was not a long-range plan, but it was just enough to get everyone moving and to determine whether we had a grasp of reality.

Common Code Base for Alpha and Integrity

The first implementation decision was to have a common code base for Alpha and Integrity. (VAX and Alpha have different code bases. We were not going to make that mistake again!) In practice, we added Integrity support to the existing Alpha sources. There were three kinds of work:

- Creating a few Integrity-specific modules
- Conditionalizing code in existing modules
- Rewriting some existing routines so that they could be executed on both platforms

Having a common code base was going to make this a very different project from that of porting from VAX to Alpha, as you will see.

External Help

Remember that we promised that “the OpenVMS Alpha schedule would remain unchanged.” We needed to do the porting work while the Alpha development work continued. Some of the new Integrity work could only be done by a specific set of OpenVMS engineers. But a much, much larger set of work would be closer to our “Recompile, Relink, and Go” direction.

EDS has supported many of the OpenVMS utilities for years and the engineers are familiar with a lot of our code, build procedures, and integration testing. Therefore, we contracted EDS to port many of the OpenVMS utilities and layered products, including some they did not already support. The EDS engineers who ported and tested a large body of OpenVMS code were critical to our ability to meet our schedules.

A number of contractors also played a vital role in meeting our early schedule demands. With their prior OpenVMS internals experience, they contributed mightily right from their first day on Integrity-specific code.

Project Dependencies and Schedule

Simply put, the direction was to write, compile, link, and execute. From the earliest stages of planning, we knew that we would have BLISS and C compilers first, and then the Intel assembler (IAS) and IMACRO a few months later; but the LINKER was far in the future. A lot of code was written and compiled before we were able to link and debug any of it. This had many ramifications in the months to follow. The LINKER would undergo as much change as any single component in OpenVMS, and much of what we did for the next year, even after we released V8.0, was dictated by the evolving capabilities of the LINKER.

The System Code Debugger (SCD) was also a distant promise. SCD is akin to DEBUG except that it is used for kernel and driver debugging. Unfortunately we would have only XDELTA and print statements for debugging for a very long time. We did not know how much this would affect our schedule. What we did know was that, without SCD, everything would take longer.

The most difficult challenge was to make a sufficiently functioning OpenVMS available to the myriad layered product engineering groups, such as DECthreads, JAVA, DECnet, TCP/IP, and DECwindows, so that they could get going with their porting work.

VAX-to-Alpha as Reference

Many engineers working on the project also worked on the VAX-to-Alpha port a dozen years ago. We tried to translate as much of that experience as we could into useful insights for the current work. There were certainly some similarities but there were also major differences. Comparisons to the VAX-to-Alpha port are inevitable. The striking difference at the technical level is what I call “broad and shallow vs. narrow and deep.”

The VAX-to-Alpha port consisted of two major challenges:

1. OpenVMS had never been ported before. How would we compile all of that MACRO-32 code and make it work in a reasonable timeframe?

2. OpenVMS was written specifically for the 32-bit VAX architecture. What would it take to make it run on a 64-bit architecture? The Alpha architecture was the least of our worries, since each Alpha system has OpenVMS-specific PALcode, which makes the system look very VAX-like. In fact, the entire project was called EVAX, or “extended VAX.”

This was the “broad and shallow” project; tons of code needed to be changed, but for the vast majority it wasn’t rocket science.

The Alpha-to-Integrity port had one gigantic challenge. Integrity doesn’t look anything like a VAX and some concepts and code at the very core of the operating systems would be completely new. Meanwhile, the vast majority of the system would require little work, if any. This was the “narrow and deep” project — it would take far fewer engineers but the time required to get from “no bits to ship” was about the same as the VAX-to-Alpha port.

Evaluation Releases

We decided to distribute one internal baselevel to a few select groups and then release two external evaluation releases before the official Field Test. The internal baselevel would be provided to the compiler groups so that they could verify that their generated code was working correctly.

The first evaluation release (E8.0) was to be a “cross tools” environment, where developers could compile and link on Alpha and run the resulting image on Integrity. It would consist of a limited set of compilers, and a large part of the operating system itself would be in place. There were restrictions, but E8.0 would give our work a little exposure and would allow a few adventurous 3rd party developers to kick the tires.

The second evaluation release (E8.1) would be released 6 months later and would provide a “native” environment where code could compile, link, and run on the Integrity system. We would add more partners as the system became more complete. We might even add another hardware platform at this stage.

These two evaluation releases were for Integrity only. It was not until the official Field Test that both Alpha and Integrity customers would see the new version of OpenVMS. (It would be the largest Field Test in OpenVMS history, with 48 Alpha installations and 54 Integrity installations.)

Development Begins

It is impossible to pinpoint a precise day on which development began. During the next few months bits and pieces of work were happening somewhat independently as the design team began to understand what they needed. At the same time, procedures were established, defining the development environment. By the end of 2001, momentum grew as a small group of engineers became connected and started moving in unison. The first release stream code change was made on January 15, 2002.

We were flattered by the confidence shown in us by The Inquirer when its headline of 1/2/2002 proclaimed:

Compaq boots VMS on Itanic

Cool. Even though this was still 13 days before our first code checkin, we enjoyed the publicity anyway!

Compilers

In the beginning, we were limited to the cross tools environment. The compilers ran on Alpha and generated code for Integrity. It would remain like that for a long time — a running OpenVMS on Integrity was at least a year away. Once again, some creative thinking proved valuable.

We would get to a running system faster if the build compilers generated reasonable code right away. The C compiler engineers built prototype Integrity Linux compilers to test the generated code and ELF/DWARF generation on Integrity Linux systems. Much of the Linux work in the compilers had

already been developed for the Alpha Linux compilers that we were selling. Starting work on the Integrity Linux compilers in September 2001, the compiler engineers used their existing compilers to test about 95% of the code generation and much of the ELF/DWARF generation that was needed for the first system boot.

An important step in early compiler development was determining which built-ins OpenVMS would need. The compiler teams looked at some existing Intel compilers and started with those built-ins, adding a few more as the operating system needs became clearer. We needed far fewer built-ins for MACRO-32 and BLISS, because new code that was specific to Integrity would be written in C.

OpenVMS has its fair share of C ASMs on Alpha. They allow:

- Referencing R26 as the caller's return address register
- Programming atomicity with LOAD-LOCKED / STORE-CONDITIONAL instructions

Both the Alpha and Integrity C compilers defined built-ins `__RETURN_ADDRESS` and `__CMP_SWAP_QUAD`, so we eliminated the ASMs and produced common code.

MACRO-32 code posed special challenges. It was easy in the VAX-to-Alpha port because the two calling standards are more or less the same – we designed them that way. We were now faced with a very different calling standard. One of the most dramatic differences is that the Intel standard defines only four preserved general registers (R4-R7), whereas Alpha defines fourteen (R2-R15). This meant that many assumptions in the MACRO-32 code were not valid for Integrity. IMACRO (the MACRO-32 compiler on Integrity) recognizes this difference and, unless specifically flagged by a new compiler directive, saves/restores registers that may be at risk. We also recoded these types of operations when convenient, especially if they were performance sensitive.

CALLs to routines that return data in registers other than R0 or R1 must also be flagged with the new directive so IMACRO will not overwrite the returned value. We devised a way for the LINKER to provide diagnostic warnings when register assumptions could not be validated. This was painful in the beginning but helped us immensely as time went on. We then defined all the linkages used by OpenVMS in a new file `$IA64_LINKAGES`, which is inserted into each MACRO-32 source module using `ARCH_DEFS.MAR`.

Another MACRO-32 problem was the hand coding of a CALL; that is, moving arguments into R16 through R21, putting the number of arguments in R25, and then JSBing to a standard CALL routine – truly evil! All these occurrences had to be changed due to the different calling standard, so we recoded them to CALLs. If the world were perfect everything would be a standard CALL, but that was not exactly on anyone's mind during the first 12 years of OpenVMS development. In a way, we had made it "too easy in VAX-to-Alpha" (a phrase we've uttered numerous times during the Alpha-to-Integrity porting project).

BLISS global registers were an unexpected challenge. A BLISS global register is one defined to have a common use for all modules in an image, such as RMS's GLOBAL REGISTER that contains the FAB. This doesn't fit into any Intel register convention. BLISS code generation was enhanced to recognize such registers with possible "live" values and protects them.

Many years ago there was a monster movie called *The Thing*. We encountered one of its descendents – Not a Thing (known as NaT). NaT is the 65th bit of a register. When set, it says that the register contents are invalid. It is used as part of speculative access. Unless you use a special instruction, you can only save a register to memory if the NaT bit is 0 (the register is not NaT). In other words, speculative access (no change) is allowed but a real change is not.

This was a new concept for IMACRO and it took awhile to work out the kinks. The operating system struggled too, because correctly preserving the NaT state when switching context turned out to be nontrivial. At last the monster was slain.

Learning Itanium Assembly Language

We had a goal of writing as little assembler code as possible but we knew there would be some. How do we get started without an assembler? It would be a few months before IAS was ported to OpenVMS; this was due to timing and schedules rather than degree of difficulty. Eventually we took the Open Source IAS assembler and added our limited ELF extensions, a relatively small job. But in the meantime we got an Itanium1 ProLiant DL590/64 and installed Linux. Reading the manuals is one thing but we really learned about 3-instruction bundles, predicates, and unwind directives by writing C code, compiling it, and studying the generated code on our Linux system.

In fact, the following were first compiled and debugged with GNU C and the assembler on Linux:

- SWIS
- EXCEPTION
- The Interruption Vector Table (IVT)
- The Translation Lookaside Buffer (TLB) miss handler

Compiling, linking, and debugging this crucial code depended on Linux tools running on Itanium hardware for some time. This allowed basic OpenVMS functions to be ready when needed by IPB and SYSBOOT.

When we originally made the decision to go with ELF/DWARF we didn't realize how much it would help us during early development. The theory of why it made sense for applications also made sense for OpenVMS.

Replacing the Alpha PALcode

There were three primary areas of work:

- Interrupts, IPL, and privilege mode management done with little OpenVMS knowledge
- Direct PAL routine calls
- Memory management

For speed and simplicity, we defined a system service for each PAL call by creating the appropriate header and library files for C, IMACRO, and BLISS. This got us going quickly and also made it easy for later performance improvements.

SWIS is at the heart of what makes OpenVMS work on Integrity. Everything in the system that requires a check for, or manipulation of, IPL and privilege mode is controlled by SWIS. The data in the SWIS log became the mainstay of debugging the most serious problems we encountered.

Emulation of the interlocked VAX queue instructions was trouble from Day 1. There are many of them and they are complicated. OpenVMS doesn't work at all without many of these instructions.

The memory management changes consisted of virtual-to-physical address translation, alignment fault handling, and PROBE emulation. We devised a scheme for maintaining Alpha virtual address translation and address space layout with GH regions using the VHPT, a basic Itanium construct.

Porting Guidelines

We wrote a short document containing coding practices. Remember that we were adding Integrity support to the Alpha source code while V7.3-2 for Alpha was in full development. The code itself was not the only thing that needed work. The development environment required serious enhancement because we would be building for Alpha and Integrity from common sources, using common building procedures, on the same cluster. Porting guidelines were a must for all facets of the project. As this small document matured it was used by other development groups, too.

You might expect "binary thinking" from computer programmers but we realized very early that some of our existing code base and build environment had a very definite "it's either VAX or Alpha"

approach when differences needed consideration. This was perfectly understandable given where we were when we ported from VAX to Alpha; however, it was short-sighted and would cause much work later. We were not going to make that mistake again.

One of our first warnings to programmers was, “watch out for those binary conditionals” and make sure they turn into something as general as possible for the future. All conditionals had to be meticulously checked and many of them needed modification. This was as true for DCL command procedures as it was for C, BLISS, etc. This was so important that as each conditional was checked the engineer added the comment “Verified for IA64 port” even if no change was needed. Here are a few examples:

- IF VAX (.....) ELSE (.....) ENDIF
- IF ALPHA (.....) ELSE (.....) ENDIF
- IF VAX (.....) ENDIF followed by IF ALPHA (.....) ENDIF

It is easy to see that all these instructions have problems when a third alternative arrives. The first two examples result in either the VAX or Alpha code being used on Integrity. This produces a compile or link error on Integrity and the programmer has to figure out the necessary adjustment. But the third is particularly troublesome because it produces, without error, no code at all on Integrity — not desirable! We definitely underestimated the amount of work that needed to be done to our existing conditionalized code. In order to maintain readable, common sources we frequently buried the conditionals in a macro.

Boot Sequence Takes Shape

In the fall of 2001 we bought our first HP i2000s. We were still part of COMPAQ but the proposed merger with HP had just recently been announced and we figured the i2000 would give us a head start in our future company.

Very shortly we reached the point of needing regular, coordinated meetings for those designing and writing code in the boot path. The first “Debug Meeting” was held on November 14, 2001, and this meeting continued multiple times a week for almost two years. They were not status meetings; they were for asking dumb questions, showing our ignorance, and especially learning from one another — and they were usually somewhat entertaining. The Debug Meetings were for developers only — no managers allowed, 30-45 minutes and get back to work. They were for down-in-the-trenches bits and bytes discussion of what was and was not working and how to keep moving forward.

In the spring of 2002 we acquired some Intel “white boxes,” generic development starters, engineering samples that are loaned to companies creating products based on Intel processors. These white boxes nearly doubled our systems. In addition, we were learning different experimentation techniques to assist us in our design and development.

For this discussion, the boot sequence is defined as the code needed to get from issuing the boot command to having loaded all of the necessary execlets (the set of images that constitute the core of OpenVMS). The code of interest is in:

- VMS_LOADER.EFI
- IPB.EXE
- SYSBOOT.EXE
- EXEC_INIT.EXE

Before describing the code we should review the generic Intel-defined boot path structure:

1. There must be a File Allocation Table (FAT) partition on the boot device. OpenVMS does not have disk “partitions” in the PC sense. The OpenVMS FAT partition is a container file; that is, a regular OpenVMS file with special contents and format on the system disk.

2. One of the files in the partition is the operating system's loader. This file is created by each operating system development group wanting to boot on an Intel Itanium architecture system.

As you can tell from the .EFI extension, the first code to execute is not an OpenVMS image. It is an Extensible Firmware Interface (EFI) application written by OpenVMS Engineering that prepares the system and then loads and starts IPB.EXE. IPB is the Itanium Primary Bootstrap, which is analogous to APB.EXE on Alpha and VMB.EXE on VAX. In addition to the boot sequence logic, the loader contains a primitive file system so that it can find, load, and start an OpenVMS image before the normal mechanisms are available. Thus, even the very first piece of code executed has to know about the inner construction of an ELF image.

The loader is completely new code for Integrity. One way to look at its function is that it gets things in place so that IPB.EXE can function much the same as its counterpart on Alpha. VMS_LOADER.EFI does many of the same things that the firmware does on Alpha, such as creating the configuration tree and the HWRPB. In turn, IPB does as much as possible to make things the same on Integrity as on Alpha for SYSBOOT.EXE. Once SYSBOOT.EXE starts, things look nearly identical to Alpha in terms of any external interfaces (for example, stopping to examine or change system parameters). SYSBOOT.EXE creates a number of data structures, loads a few execlets, and finally transfers control to EXEC_INIT.EXE.

At this point, most of the platform-specific code in the boot sequence is complete. EXEC_INIT.EXE loads the rest of the execlets (typically a couple of dozen), executes their initialization routines, and transfers control to the SWAPPER process.

Another project theme was "don't be different" without a very compelling reason. This challenged us in a number of areas but none more so than ACPI with its relationship to the boot path. One manifestation of "don't be different" was if other HP Integrity operating systems did something strictly by the Intel specifications, then OpenVMS should too. We went out of our way not to require special treatment from the architecture or the firmware designers or implementers. Booting the system is one of those areas where operating systems can vary greatly. Specifically, most operating systems consist of one monolithic file that gets loaded. Knowing that any relevant code is already running, ACPI then provides all the configuration data. This is clearly not the OpenVMS model you have seen in the previous paragraphs.

Primitive Debugging

Before we had the luxury of a debugger or the ability to take a crash dump, various forms of print statements were the only means of tracking down problems. We suffered with print statements for awhile, still making progress. Then one of those silver bullets materialized – the instruction decoder. The callable instruction decoder and instruction format structure definitions are the foundation for all of the debugging tools, like DELTA, XDELTA, DEBUG, SDA, SCD, and PCA. In addition, the format definitions are used by the base operating system during alignment fault handling. When XDELTA became functional, breakpoints and single stepping instructions emerged to increase our productivity. While most of the underlying debugging code was completely new for Integrity, there was one thing that was "old." Setting the PSR.ss bit is just like the Tbit on VAX, and much simpler than the way it's done on Alpha.

Being able to take a crash dump is a momentous event in a project like this and our first one happened quite unintentionally. While stepping through some code, a breakpoint was incorrectly positioned. A ";P" caused an ACCVIO and, lo and behold, a crash dump was executed perfectly! The code had been recently compiled and linked and was just waiting to be exercised. It was one of unusual moments where seeing a crash happen is a joyous occasion.

Early "Builds"

During the first half of 2002 a significant amount of code was written and the compilers began to take shape. A few OpenVMS source code facilities were in the early stages of building for Integrity. It was way too soon to think about doing a full Integrity build, so we concentrated on the few pieces we

needed to continue making progress. In the early days we built only those few images and then copied them to an OpenVMS Alpha system disk, using that to test the early stages of booting. This was a little cumbersome but it required only a few engineers and it moved us along quite well. One of the nice things about using industry standard components was that whenever we needed a few new disks we just went over to CompUSA and picked up a few — a very new concept for OpenVMS development.

At this point in the project we did not yet have the LIBRARIAN, which is at the heart of how we build most of OpenVMS. We changed the build process so that it does not create the big object library and delete the .OBjs, and then we changed all of the link procedures to include the .OBjs directly rather than extracting them from the library.

Our LINKER capabilities were so limited at the time that OpenVMS was forced to run in a dramatically unusual way. The LINKER could link a single image, but it could not yet link executables that call from one to another through the base image. Everyone who was around OpenVMS 20 years ago (prior to V5.0) will remember the monolithic SYS.EXE. That was not quite what we needed now but a very, very large SYSBOOT.EXE did fill the bill. During the very early days of debugging the growing boot path, all the code that we needed that normally resides in the executables was linked into SYSBOOT.EXE. We didn't need this throwback solution for very long, but every single day counted and it kept us moving.

The first general Integrity build was announced on June 3, 2002. It was really the starting point for everyone who was not working on the boot path to see the result of compiling their code. (Loosely translated, this meant discovering just how well MACRO-32 code survived the new IMACRO compiler.)

Now part of HP, we received our first internal shipment of rx2600s in July of 2002. The project was starting to look almost normal — regular builds, regular meetings, a growing number of development systems, and some code that worked. With the i2000s, the white boxes, and now the rx2600s, every developer doing significant work had a system for exclusive use. As the number of implementers grew, so did our need for more and better procedures.

Since we had no networking capability on Integrity yet, we needed a way to quickly and easily move files from our development cluster to the Integrity systems. We took an HP disk system 2100 (a "pizza box" or "pancake") and connected it to a small OpenVMS Alpha system and six Integrity systems, replicating this setup numerous times. This is a completely unsupported configuration, of course, but with careful mounting and dismounting it works. We would build on the development cluster and copy the resulting kit to the "server" Alphas. Individual developers did system disk builds on their designated disk, dismounted it, and booted their Integrity system.

It was about this time that we were starting to realize how much larger the image size on Integrity was, due to relocation data, unwind data, and the number of instructions. Added to that was the traceback information that we had decided to create, to help with debugging in the field. The result is that OpenVMS images on Integrity (application software as well as the operating system images) are on the order of three times larger than the corresponding image on Alpha.

Boot Contest

The "First Boot" is always a milestone in a project like this. From the technology standpoint, however, it is only a minor proof point that the final goal might be attainable. So much under-the-hood cheating is going on that it borders on fantasy. For the most part, people understand this. Nevertheless, given the building anticipation, it is still extremely exciting.

The first boot of OpenVMS on Integrity became a public contest. Predict exactly when it would happen, and win a prize! (This just added to the pressure when we were struggling to get anything at all to work, let alone do something as identifiable as a system boot.)

Every contest must have rules. Here was the official definition of "first boot:"

Porting OpenVMS to HP Integrity Servers - Clair Grant

1. System must boot "MIN."
2. Must be able to login as SYSTEM.
3. \$ DIRECTORY returns the correct list of files.

There was no requirement for security, network, clustering, SMP, batch, editor, or anything else not absolutely necessary.

In estimating the expected date for the project plan, we looked back to the VAX-to-Alpha port, when we booted OpenVMS six months after we had reasonably reliable, linked images. Back then we were working on hardware that was in development. For the Alpha-to-Integrity first boot, given that we were using Integrity systems that were shipping HP products, we reasoned that it might take half as long, so we established the goal of three months, which translated into February 13, 2003. Diligent engineering plans notwithstanding, someone announced to the world that we were trying to boot by the end of 2002!

Not All Fantasy

Many fundamental pieces of the system must be working in order to achieve even a minimalist first boot. Here are some:

- IMACRO-32, BLISS, and C compilers
- LINKER
- Device discovery
- Memory layout
- Device driver
- Platform support
- Interrupts
- Clock ticks
- Memory management
- Lowering/raising IPL
- AST delivery
- MOUNT
- File system
- RMS
- DCL
- Image activator

Within these parameters, cheating was allowed, even encouraged. Exception handling, error returns, security, accounting – who cares, as long as the DIRECTORY listing is correct.

The official time of the first boot of OpenVMS on Integrity was January 31, 2003 at 3:31 PM EST on an HP i2000 (Itanium1) system. This prompted hugs, handshakes, and congratulatory words. It was a huge boost to our confidence and a gigantic sense of relief! The verified Boot Contest directory output and the winners of the contest are in Appendix D – Boot Contest is Official: January 31, 2003 at 3:31 PM EST.

NOTE: After listing the files we didn't return to the \$ prompt. The rules of the contest specifically left that out (in case anyone is getting picky). Why? The last event in processing the DIRECTORY

Porting OpenVMS to HP Integrity Servers - Clair Grant

command is to return "file not found." Since the system was not even close to handling signals or errors, it stopped at a breakpoint after printing out the last file specification!

The complete list of files used in the first boot is in Appendix E - Boot Contest Code.

Internal Baselevel – March 28, 2003

OpenVMS booted on an HP rx2600 (Itanium2) system on March 17, 2003; this was the platform that most of our early users would have. Less than two weeks later on March 28, 2003 we released our first internal baselevel to the compiler groups. We were definitely rolling. Having a system we were confident enough to give to other groups expecting to do their own development and testing was a gigantic step. Soon we had DECnet and TCP/IP, making it easier to share development systems.

We first booted an Integrity system into a cluster with Alpha and VAX systems on May 15, 2003. The first public clustered demo was May 19, 2003 at the HP-Interex Conference in Amsterdam. See Appendix F - The Cluster T-Shirt for the \$SHOW CLUSTER command output from our development lab cluster. (The cluster contained a VAX but it was edited out of the picture due to early uncertainty concerning our support statement for VAX and Integrity in the same cluster.)

There was still a long way to go, but booting networked, clustered systems confirmed our feeling that a completely functioning OpenVMS on Integrity was possible. When we started, many people were extremely skeptical that we would ever match the quality and performance of OpenVMS on Alpha. Others said it would be impossible to make it work at all. It should be noted that, 12 years earlier, there was the same skepticism (05%) about porting from VAX to Alpha. The 1990 nay-sayers had more credibility, since it was the very first attempt at porting OpenVMS and yes, it was indeed questionable as to whether it could be done. This time we never had any question about being able to do the port. The only question was how well the system would perform.

OpenVMS V8.0 – June 30, 2003

The internal code name for V8.0 was MAKO. The development environment used cross tools. We compiled and linked on Alpha and moved the images to an Integrity system. The goal of this first evaluation release was to give some of our 3rd party providers the opportunity to get an early start. We hoped that they would be ready when our first general release shipped. Also, it would give us some limited exposure in an environment other than our own. Things were starting to get serious.

The cross tools were:

- Compilers – BLISS32, BLISS64, C, IMACRO, FORTRAN
- IAS assembler
- LINKER
- CRFSHR
- LIBRARIAN
- CHECKSUM
- CDU
- ANALYZE_OBJ

OpenVMS Engineering created a special program called Fast Track and provided dedicated support for 20 selected partners, albeit with a very limited OpenVMS environment, while we learned just how things were starting to shape up.

A Technical Exchange in Nashua on July 8-9, 2003 with engineers from HP firmware and operating system groups, in particular one of the HP team members who co-developed the Itanium architecture with Intel, was enlightening. Although we had a functioning system, it was still one of those "the more you know, the more you realize you don't know" experiences.

Porting OpenVMS to HP Integrity Servers - Clair Grant

Now more OpenVMS engineers began to get involved with the compiling and linking of various components of the system. We held a series of lunch-time seminars open to the entire engineering community. The engineers who had been doing the initial development work presented details of the new Integrity-specific areas of the system.

What's in a Name?

In 1998 at a very public forum in Los Angeles, OpenVMS Business Management said, "There will never be a V8.0 of OpenVMS." It would be V7.x forever, implying that we would be causing as little perturbation as possible in the system for the rest of its lifetime. At the time, there was a different mindset regarding the future of OpenVMS.

Times had changed. In 2003 there was widespread debate within OpenVMS as to which OpenVMS Integrity release would be dubbed V8.0. "V8" became the local catch-phrase, and large quantities of a certain juice maker's product mysteriously appeared in conspicuous places.

OpenVMS V8.1 – December 18, 2003

The internal code name for V8.1 was JAWS and it came from a native development environment. We compiled and linked on the Integrity system. The goal of this second evaluation release was twofold:

- Increase the number of partners.
- Establish a complete Integrity environment.

Compiling and linking was now supported on the Integrity systems – no more cross tools and transferring files. In fact, one of the early tests of this environment was building the tools themselves natively (for example, a compiler compiling itself).

Whole Program Floating Point Mode

Floating Point...floating point...floating point! Floating point works very differently on Alpha and Itanium. Source code modules that have been compiled with different floating point qualifiers (control, rounding, and precision), as well as the objects, get linked into a single image. On Alpha, floating point preferences are part of the instruction stream generated by the compilers. On Integrity, they are defined by the contents of the Floating Point Status Register (FPSR) at the time of instruction execution.

On Alpha, you get exactly what you declared for each module with no decisions or assumptions needed along the way. On Integrity, the floating point preferences are noted in each object file and the LINKER stores in the image the floating point preferences of the module with the MAIN entry point. After an image is loaded into memory but before any instructions are executed, the FPSR is set with the values established by the LINKER. As you can see, there is no way for OpenVMS to magically reproduce the exact Alpha behavior for this mixed-mode case. For an application that is sensitive to the individual module declarations, there are system services, namely SYS\$IEEE_SET_*, for dynamic changes as the program executes.

Converting from VAX F/D/G to IEEE

We had to add more library routines to help applications convert from VAX floating point types to IEEE. Many routines already existed but now we were actively encouraging application writers to convert and we discovered that there were missing functions. Who would do this unplanned work at this very late date? We enlisted some engineers at the Customer Support Center and they ably produced the desired results.

Locking the Working Set in Memory

Privileged processes, which enter kernel mode and raise IPL, lock their code and data in the working set so that the pagefault handler doesn't get invoked at too high an IPL for I/O. The system service SYS\$LKWSET accepts a range of addresses and locks the appropriate pages in memory. For

compatibility, we changed SYS\$LKWSET to lock the entire image in memory when an address in the input range is found to be in the code image section.

Only the code and data need to be locked on VAX, but Alpha locks code, data, and linkage data – a little messier but doable. Because of the new ELF image format on Integrity, programmers have to lock code, data, short data, and linkage data, which is nearly impossible to figure out at the application level. We highly recommend that applications use LIB\$LOCK_IMAGE to lock the entire image in memory; it is easier to use than multiple calls to SYS\$LKWSET. This new \$LIB routine is available on both Alpha and Integrity, so you can maintain common code. The entire image is locked, so the process may require larger working set quotas. In OpenVMS, we switched many of our privileged processes to use LIB\$LOCK_IMAGE and increased the SYSTEM account's working set quota.

OpenVMS V8.2

Superdome Demo

There is nothing like a little publicity. The Superdome 4-OS (HP-UX, Linux, OpenVMS, Windows) demo at the HP Analysts Meeting in Westboro, MA. on February 13, 2004 drew lots of attention. The 4-cell system was configured as 4 nPars (hard partitions), each running a different operating system.

V8.2 does not support cell-based systems, which presented some new problems to overcome. Since this was a demo (some cheating allowed) we hard-coded a few I/O-related addresses in the drivers rather than devise a complete solution. We even had to boot OpenVMS in cell 0, but these quick, temporary measures brought a high payback.

50-bit Physical Addressing

A lightning bolt struck at the wrong time. Everything was in place for the first official Integrity Field Test in mid-summer 2004, but when we started testing with the mx2 CPUs for the rx4640 we had a revelation: the mx2 cache brought with it the need to use addresses beyond the OpenVMS-supported 43-bit maximum. One option was to not support the 8-CPU version of the rx4640 and postpone the necessary memory management work until the next release, but we decided to do the work in V8.2 because additional new platforms might come along before then. The data structure changes and source code changes would affect any privileged code that worked directly with PFNs, for example DECnet and TCP/IP, and we wanted to get all known, mandatory privileged code changes into the first Integrity release.

A design was created, a project plan written, and five engineers were drafted on very short notice to make the OpenVMS changes:

1. Expand the PFN field in the PTE from 32 bits to 40 bits.
2. Expand all other PFN fields in data structures, global cells, and routine interfaces to 64 bits.

These changes were extensive and for Integrity only; however, common source code was maintained using macros in which the platform differences were buried. This work disrupted the weekly builds for awhile and the affected layered products (who had thought they were already done) now had difficult work to do – not a lot, but in sensitive areas. As the base operating system work concluded, the 50-Bit Physical Addressing Programming Cookbook helped developers complete the extra work we had foisted on them at the last minute and everyone survived.

Rewriting the VAX Queue Instruction Emulation

Replacing the Alpha PAL calls with OpenVMS system services worked well in most cases. Early in the project we created the "promote page" as our mechanism for employing Integrity's Execute Privileged Code (EPC) instruction. We used this mechanism for entering kernel mode before transferring to SWIS to switch stacks, save registers, and call the system service dispatcher.

Porting OpenVMS to HP Integrity Servers - Clair Grant

EPC, without SWIS, switches privilege modes without switching stacks, saving registers, or dealing with IPLs, ASTs, etc. This is exactly what we needed for the performance-critical queue instruction emulation. However, without SWIS to set up stacks, there are many restrictions. The code had to be written carefully in assembly language, but we were able, after a few misadventures, to create exactly the code we needed.

Checking access to possibly misaligned queue elements in threaded process environments was difficult. We programmed exception handlers to catch queue elements that the caller's privilege mode cannot access. We also designed a clever use of the TAK, PROBER, and PROBEW instructions with interrupts disabled to quickly determine if the code could proceed to atomically execute the queue instruction. After a few false starts, we finally had a good solution. EPC gave us just what we needed. A few other PAL calls have been changed to use this method.

The Final V8.2 Changes

Just before Christmas 2004 the irrepressible NaT consumption fault made another appearance. It was found and fixed and we went home for the holidays thinking that we were done. The test clusters would run over the holiday break and we would claim victory when we returned. Unfortunately, the results were not satisfactory. In fact, during the holidays a few of the engineers noticed a couple of problems.

Floating Point...Floating Point...Floating Point!

There are 128 floating point registers on Integrity systems. It would be desirable to not save/restore all these registers unnecessarily when context is changed. To that end they are architecturally treated as two sets, F0-F31 (low) and F32-F127 (high). Status bits, maintained by the processor for each set, indicate whether a change has been made to a register in the set. In many cases, the high floating point register set is not used and OpenVMS takes advantage of the "modified" bit indicators to eliminate unnecessary saves/restores. There was a case involving AST delivery where we didn't have it quite right and it took us a very long time to find it.

Then we encountered another floating point problem! Floating point register corruption was happening when correctable memory errors were reported. Not only was this problem difficult to find, but it was involved with our internal OpenVMS fiat about floating point usage. OpenVMS and HP-UX have an agreement with the compiler writers that if code is compiled using `/INSTRUCTION_SET=NOFLOAT`, only registers F6-F11 can be used in the generated code. The operating system is compiled this way in order to limit the saves and restores at interrupt level. The register corruption happened at the same time as the memory errors. Firmware test engineering provided a test program that inserts correctable memory errors. We quickly ported it to OpenVMS, increased the error polling rate, then diagnosed and fixed the problem. The OpenVMS error logger was calling firmware, and not appropriately preserving the floating point registers.

The last code change in the V8.2 release was made on January 10, 2005 and the final test runs were started. They ran with no problems. We had finally made it! The OpenVMS Engineering Readiness Review was held on January 13, 2005 and it was official – the final DVD was sent to manufacturing.



Performance Evaluation

Now that we had a production-quality system, how well did it perform? That is a very open-ended question so we decided to concentrate on how comparable Alpha and Integrity hardware platforms performed with a selected set of CPU, memory, and I/O tests. For example, we compared an Alpha DS25 with an Integrity rx2600 and an Alpha ES45 with an Integrity rx4640.

In general, these Alpha and Integrity systems perform about the same. In some tests, one is slightly better than the other, and different tests produce results that are reversed, but in most cases the differences are small. When "the announcement" occurred way back in 2001 it was stated that the performance leadership crossover from Alpha leadership to Integrity would be roughly 2005. That turned out to be a very accurate prediction.

Integrity systems have a wealth of performance counters accessible from software. OpenVMS tools have incorporated the ability to gather the data and relate appropriate information to instruction pointers in routines. This helped in tracking down and improving some initially poorly performing areas, for example needless stack unwinding (very expensive) during certain calling sequences of an RTL. Another example was an inefficient OTS data-moving routine.

One universal finding was that everyone had to eliminate unaligned data. An unaligned data reference on Alpha is expensive, but it can be two orders of magnitude worse on Integrity. So be sure to naturally align those data cells!

As we gain more experience on these new platforms there will certainly be on-going operating system and compiler performance improvements, as well as CPU and platform speed-ups. This will be true for the rx1600, rx2600, and rx4640 (supported by V8.2), and even more so for the larger systems supported by future releases of OpenVMS.

Summary

Three and a half years of work resulted in OpenVMS I64 on Integrity servers. We tried to make moving from Alpha to Integrity as easy as possible for the application providers. We know there will be an occasional exception but for the vast majority of applications we achieved our goal. We may have caused minor inconveniences when it was not strictly necessary for Integrity, but part of the goal was to make OpenVMS a better system in 2005 than it was in 2001, and that's what we did.

A project of this magnitude happens only a few times during the life of a product. This is the second time for OpenVMS and it is still moving forward.

Acknowledgements

Porting OpenVMS encompassed the work of more people and groups than can be adequately recognized here. The entire OpenVMS community participated in this enormous undertaking.

Porting OpenVMS to HP Integrity Servers - Clair Grant

Everyone was dedicated to the highest standards in producing a quality product worthy of our customers.

This port to Integrity was the result of the efforts of many very talented people, but the work done by those who contributed to OpenVMS in previous years, even previous decades, also contributed greatly. OpenVMS is a great system to work with in terms of concept and implementation, and we are indebted to all of those who came before us.

A special "thank you" goes to the reviewers of this article – Burns Fisher, Karen Noel, John Reagan, Ron Brender, Jeff Nelson, Josh Cope, Gaitan D'Antoni, Sue Lewis, Ken Moreau, and Randy Barth.

For more information

Send questions and comments about this article to clair.grant@hp.com.

Appendix A - How to Make Your Head Hurt

How to Make Your Head Hurt!

<u>Bytes</u>	<u>Intel®</u>	<u>Alpha</u>
1	<i>byte</i>	<i>byte</i>
2	halfword	<i>word</i>
4	<i>word</i>	longword
8	doubleword	<i>quadword</i>
16	<i>quadword</i>	octaword

© 2002 hp hp enterprise technical symposium page 1

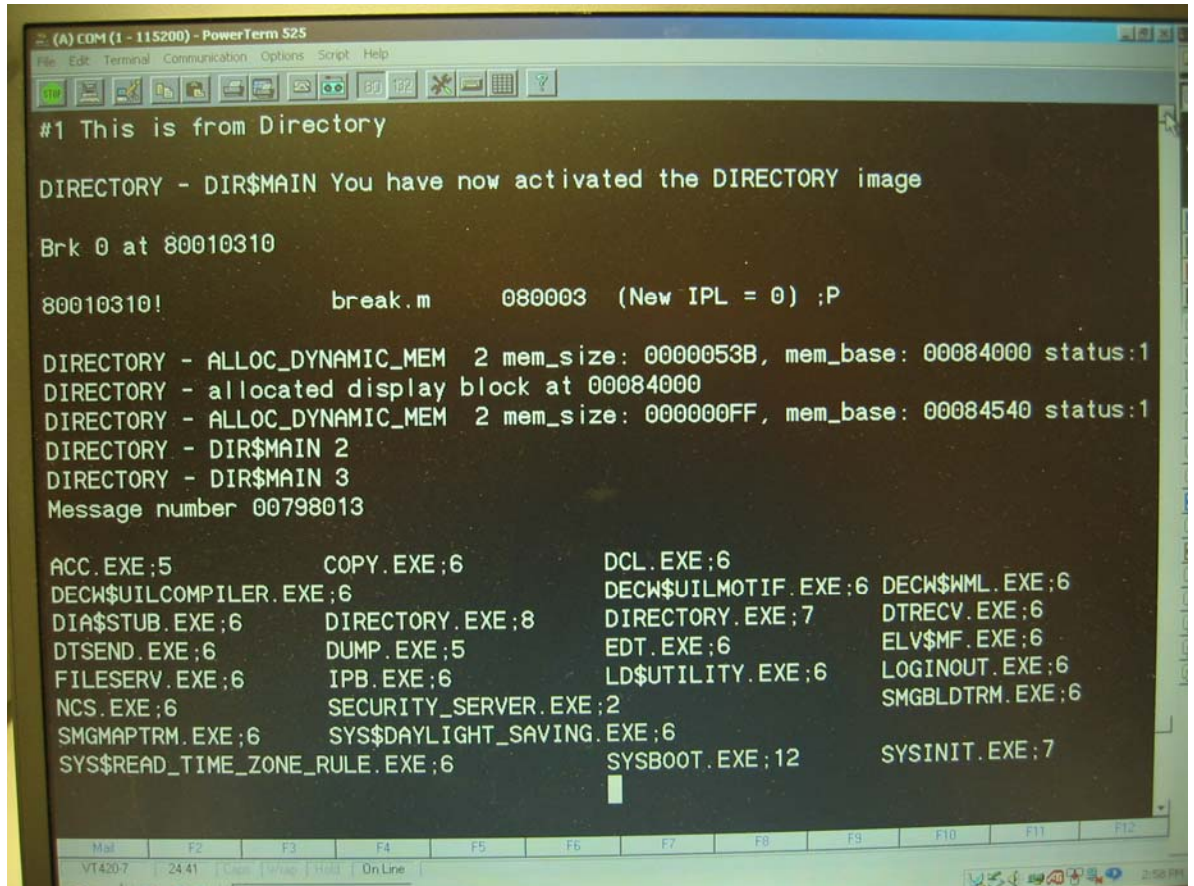
Appendix B - OpenVMS Alpha to Itanium Register Mapping

Register Mapping

Alpha : IPF	Alpha : IPF
return info { 0 = 8 } return info { 16 = 14	
1 = 9 } return info { 17 = 15	
2 = 28 } args { 18 = 16	
3 = 3 } args { 19 = 17	
4 = 4 } args { 20 = 18	
5 = 5 } args { 21 = 19	
6 = 6 } preserved { 22 = 22	
7 = 7 } preserved { 23 = 23	
8 = 26 } preserved { 24 = 24	
preserved { 9 = 27 } (AI) 25 = 25 (AI)	
10 = 10 } (RA) 26 = .. no mapping	
11 = 11 } (PV) 27 = .. no mapping	
12 = 30 } 28 = .. no mapping	
13 = 31 } (FP) 29 = 29	
14 = 20 } (SP) 30 = 12 (SP)	
15 = 21 } (RZ/sink) 31 = 0 (RZ)	

March 21, 2005
page 1

Appendix D – Boot Contest is Official: January 31, 2003 at 3:31 PM EST



The screenshot shows a terminal window titled "(A) COM (1 - 115200) - PowerTerm 525". The output text is as follows:

```
#1 This is from Directory
DIRECTORY - DIR$MAIN You have now activated the DIRECTORY image
Brk 0 at 80010310
80010310!          break.m          080003  (New IPL = 0) ;P
DIRECTORY - ALLOC_DYNAMIC_MEM 2 mem_size: 0000053B, mem_base: 00084000 status:1
DIRECTORY - allocated display block at 00084000
DIRECTORY - ALLOC_DYNAMIC_MEM 2 mem_size: 000000FF, mem_base: 00084540 status:1
DIRECTORY - DIR$MAIN 2
DIRECTORY - DIR$MAIN 3
Message number 00798013
ACC.EXE;5          COPY.EXE;6          DCL.EXE;6
DECW$UILCOMPILER.EXE;6  DECW$UILMOTIF.EXE;6  DECW$WML.EXE;6
DIA$STUB.EXE;6      DIRECTORY.EXE;8      DIRECTORY.EXE;7      DTRECV.EXE;6
DTSEND.EXE;6        DUMP.EXE;5          EDT.EXE;6            ELV$MF.EXE;6
FILESERV.EXE;6      IPB.EXE;6           LD$UTILITY.EXE;6     LOGINOUT.EXE;6
NCS.EXE;6           SECURITY_SERVER.EXE;2  SMGBLDTRM.EXE;6
SMGMAPTRM.EXE;6     SYS$DAYLIGHT_SAVING.EXE;6  SYSBOOT.EXE;12      SYSINIT.EXE;7
SYS$READ_TIME_ZONE_RULE.EXE;6
```

Winners of the Boot Contest shirts were:

- Luciana Silva
- Gabriel Sterk
- Randy Loch
- Geoff Bryant
- Kurt Huddleston
- James Wilkinson
- Lee Mah
- Alan Frisbie
- Hans Vlems

Appendix E - Boot Contest Code

These files contain the 1361 modules that completed the first boot of OpenVMS on Integrity.

vms_loader.efi	sys\$dqbtddriver.exe
ipb.exe	system_primitives_min.exe
sysboot.exe	system_synchronization_uni.exe
sys\$public_vectors.exe	errorlog.exe
sys\$base_image.exe	exec_init.exe
sys\$cpu_routines_4001.exe	system_debug.exe
sys\$srbtddriver.exe	ia64vmssys.par
sys\$opddriver.exe	
	logical_names.exe
exception.exe	sys\$ttddriver.exe
io_routines.exe	sys\$srddriver.exe
sysdevice.exe	sys\$dqddriver.exe
process_management.exe	vms\$system_images.dat
sys\$vm.exe	sys\$config.dat
shell8k.exe	
sysinit.exe	
libots.exe	
librti.exe	
decc\$shr.exe	
cma\$tis_shr.exe	
image_management.exe	
locking.exe	
sysldr_dyn.exe	
security.exe	
vms_extension.exe	
f11bxqp.exe	
rms.exe	
loginout.exe	

dcl.exe
dcltables.exe
directory.exe
util\$share.exe
directmsg
sysmsg

Appendix F - The Cluster T-Shirt

\$ SHOW CLUSTER 

View of Cluster from system ID 20458 node: CONMAN 6-JUN-2003 09:44:15

NODE	HW_TYPE	SOFTWARE	STATUS
CONMAN	HP rx2600	VMS X9TQ	MEMBER
IAVMS3	HP zx2000	VMS X9TQ	MEMBER
CLU24	COMPAQ Professional Workstatio	VMS V7.3	MEMBER
KNOTS	AlphaServer 1000A 5/300	VMS V7.3	MEMBER
TONYL	Intel W460GXBS	VMS X9TQ	MEMBER

page 1

Appendix G - Evaluation Release V8.0



© 2005 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

