Porting to the Intel IA-64 Architecture

What You Can Do Now

INTRODUCTION

The IA-64 architecture was designed to overcome the performance limitations of today's architectures and provide maximum headroom for the future. To achieve this, IA-64 offers an array of innovative features to extract greater instruction-level parallelism. Chief among them are speculation, predication, large register files, a register stack, and advanced branch architecture. Another IA-64 innovation, 64-bit memory addressability, targets the growing memory-footprint requirements of data warehousing, e-business, and other high-performance server and workstation applications. The IA-64 also provides an enhanced system architecture supporting fast interrupt response and a flexible large virtual address mode.

You may be already aware of the benefits that the Intel IA-64 architecture can bring to your application in terms of performance, scalability, and reliability. You may be planning to port your application to the IA-64 architecture as soon as you have access to the hardware. But did you know that you don't have to wait until then? That you can launch your porting effort today? In this paper, you'll learn why it's a good idea to begin porting now, how you can get started, and where you can go to find more information.

WHY PORT NOW

Just as there are compelling reasons to port your application to the IA-64 architecture, there are equally compelling reasons to begin that porting effort *today*.

- By starting your port now, you can you can finish all the coding work now and position your company to begin testing, optimization, and benchmarking as soon as you gain access to the IA-64 hardware. This means you'll be able to launch your IA-64 application that much sooner.
- By starting your port now, you can identify tool and library dependencies in time to work with third-party suppliers to address those dependencies well before hardware launch. In many cases, suppliers already have ported their software to the IA-64 architecture, but those that have not yet done so need to start now.
- By starting your port now, you can maximize your own engineering efforts by combining code changes for the IA-64 with changes you may already be making for the latest 32-bit version of your product. For example, you can merge your IA-64 porting effort with the migration of your IA-32 application to a new version of your target operating system. In so doing, you not only maximize your engineering resources but also may discover new features you want to add.

Remember also that plenty of porting tools are now available from the various operating-system vendors. That means there's nothing standing in the way of launching your IA-64 port right now. (See Appendix 1 for more information on the tools available now.)

STEP BY STEP

Consider the following basic steps required in migrating your application to a new architecture:

1. Assessing complexity
2. Developing a plan *These steps need no hardware.*

3. Modifying your code

4. Testing and validation
5. Optimization *These steps need hardware.*
6. Benchmarking

Pay particular attention to steps 1–3. Because these steps do not require access to the target hardware, you can begin them right now. In more detail, here is a rough outline of how you might want to proceed:

Assess complexity by identifying crucial third-party dependencies. From the outset you should ensure that ports are complete or at least underway for all third-party dependencies in your application. Among others, these include development tools and environments, source-code control mechanisms, static libraries, and, most important, dynamic libraries.

What makes dynamic libraries so crucial is that they provide the primary mechanism for linking binaries. This means that an IA-32 library that will continue to be accessed exclusively by IA-32 processes may remain unported, but those that are likely to be accessed by one or more IA-64 processes should by all means be ported to IA-64. This is because most operating systems do not support the mixing of 32-bit and 64-bit code within the same process, even though the IA-64 architecture allows it.

If the IA-32 library that needs porting was supplied by a third party (which is common in the Win64 environment) and the source code is not available, you need to ask the supplier to perform the port. In a worst-case scenario, where this approach is not feasible, you can use interprocess communication (IPC) to address the problem, as detailed in Appendix 2.

Develop a plan, focusing on your operating system, porting approach, training, and tools. Based on the fact that each operating system supports a slightly different porting approach, start evaluating and selecting the approach that best suits your environment. Meanwhile, begin training your engineers on IA-64 architecture and on porting and development tools available from the operating-system vendors and tool suppliers. As of this writing, numerous Intel documentation materials are available. You also can obtain the migration tools and porting guides you need from the various operating-system vendors. (See Appendix 1 for more on training tools.)

Modify your code. Use the tools available from the operating-system vendors to examine and modify your code base for the IA-64 architecture, taking particular care to address challenges that arise when you move from one data model to another. Begin by compiling the application using the vendor's software developers' kit. Address warnings and errors until all your object modules compile and link successfully under *both* 64-bit and 32-bit environments. Once this occurs, you have completed the work that can be done without access to hardware. This means you are ready to build the application and begin debugging it as soon as hardware is available.

PORTING APPROACHES

The operating-system vendors support various approaches to porting your 32-bit application to the IA-64 architecture:

Full port using 64-bit pointers and 2 gigabytes or more of virtual address space. This approach, which returns a 64-bit binary, is best when you need the performance advantages of the IA-64 and a large address space.

Small-address-space port using 64-bit pointers and up to 2 gigabytes of virtual address space. This approach, which returns a 64-bit binary, is best when you want to minimize the volume of testing required and need no more than 2 gigabytes of virtual address space. Note that this approach is available only on the Win64* platform.

Small-address-space port using 32-bit pointers and up to 2 gigabytes of virtual address space. This approach, which returns a 32-bit binary, is best when you want to minimize data expansion or "code bloat" and need no more than 2 gigabytes of virtual address space. Note that under this approach you still need to use 64-bit pointers for interaction with the operating system.

Unmodified IA-32 binaries. Note that this approach is not an actual port and so does not take advantage of IA-64 performance or expand the size of binary objects and other structures. To be precise, the performance of the IA-32 binaries will be roughly one generation behind that of existing 32-bit binaries. Consequently, this approach is most suitable when you want to provide investment protection (compatibility) for applications developed in the IA-32 environment but do not need increased performance.

CHANGING DATA MODELS

During your port to the IA-64 architecture you may face challenges, particularly as a result of moving from one data model to another. Good programming practices can help to keep such challenges to a minimum, but even if they occur just rarely they can cause problems.

Data model	Used in which OS(s)	`Int`	`Long`	`Pointer`
ILP32	All 32-bit	32 bits	32 bits	32 bits
P64	Win64	32 bits	32 bits	64 bits
LP64	All 64-bit Unix	32 bits	64 bits	64 bits

Table 1: C data models, types, and sizes as implemented on the IA-64 architecture.

To understand the nature of these challenges, consider the data types used by the C programming language and their sizes as implemented in the major operating systems, as shown in Table 1. Now consider the problems that can result from the inconsistencies in those sizes:

Data type `int` and data type `long` are different sizes (Unix only).

- When the value of a variable of type `long` is assigned to a variable of type `int`, the value is truncated.

```
int I;
long L1, L2;
I = L1 + L2;
```

- When the programmer has explicitly cast a value in the original (IA-32) source code, the value is truncated.

```
int int1, r1, r2, r3;
long long1;
void f(void)
{
r1 = long1/int1;
r2 = (int)long1/int1; /*32-bit expression → 32-
bit*/

r3 = (int)(long1/int1); /*64-bit expression → 32-
bit*/
}
```

- The incompatibility between a pointer of type `int` and a pointer of type `long` results in incorrect assessment.
- When function calls lack prototyped function declarations, the default return type is `int`. This can result in a truncated value being returned to the calling function if the calling function is expecting type `long`.
- When untyped integral constants are `int` by default, incorrect assignment results, for example in the use of `long` 1 = 2L << 32 instead of `long` l = 2 << 32.

Data type `int` and data type `pointer` are different sizes (all the major operating systems).

When a value of type `pointer` is converted to a value of type `int`, the value is truncated.

- In an arithmetic context, assuming that data values of type `int` and of type `pointer` are the same size would produce incorrect results. This is because a 32-bit integer is incremented by four bytes and a 64-bit pointer is incremented by eight bytes.
- In the absence of a function prototype, the default return type is `int`. This can result in an integer being assigned to a pointer if the calling function expects type `pointer`.

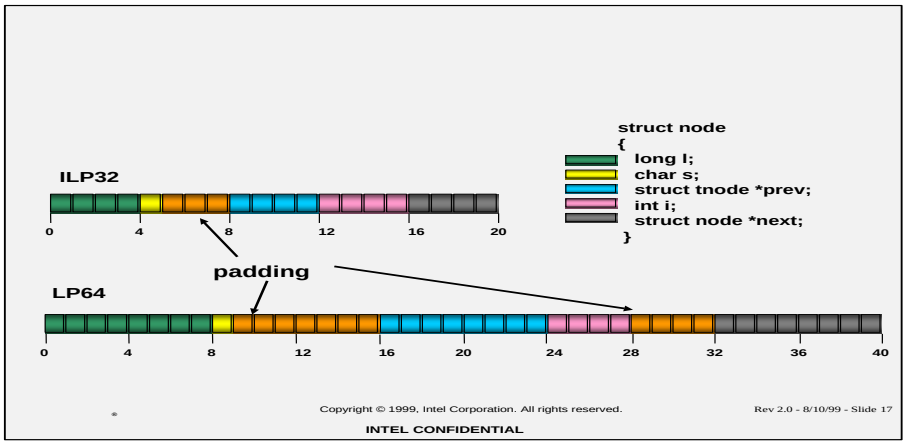

Figure 1: Pointer sizes and alignment can lead to problems with the sharing of binary data through IPC, network, or disk.

Data type `long` and data type `pointer` are 64 bits in size and conform to a 64-bit alignment (all the major operating systems).

- As shown in Figure 1, structures defined on the IA-64 architecture grow as a result of data-type size changes and data-type alignment restrictions. The alignment restriction is that structures of type long in the LP64 data model and of type pointer in all data models must be aligned on 64-bit boundaries. This may require padding, which can lead to problems with the sharing of binary data through IPC, network, or disk, especially when the alignment is not taken into consideration.

OTHER CHALLENGES

In addition to the problems caused by inconsistencies or incompatibilities in different data models, developers porting to IA-64 may encounter one or more of the following other challenges:

Undocumented/reserved bit fields in hardware or undocumented system calls in the operating system. These bit fields and system calls are undocumented or reserved for a reason and will probably change with any hardware or operating-system migration. If any code in the IA-32 version of your application uses those fields or system calls, you should consider removing or rewriting the code before porting it to the IA-64.

Unguarded `ifdefs`. When Windows upgraded from a 16-bit to a 32-bit architecture, many application vendors responded by wrapping Win32 code in `#ifdef Win32` sections and applying a `#else` statement to Win16 code. If your company is one of these vendors, you need to review these `#ifdef` statements in order to prevent the code from defaulting into the `#else` case reserved for a 16-bit operating system or hardware configuration. Note that this example is relevant to cross-platform code in any environment, not just Windows.

In-line assembly code. In-line assembly code is not supported by IA-64 compilers. If you want to use assembly code for optimizing performance, you should place it in a

file separate from that of your main code. You also should note that the primary performance gains achievable by a port to the IA-64 architecture are a consequence of compiler enhancements. This makes it a good idea to let the compiler handle optimizations whenever possible.

Self-modifying code. IA-64 binary code is significantly different from IA-32 binary code. For this reason, you should rewrite any self-modifying code so that it creates binaries of the correct format.

Data-packing. You may have portions of existing code that use data-packing to save space when writing to disk, but in a 64-bit environment this approach can create problems. Because the IA-64 architecture requires 64-bit alignment of values of type `pointer` in all data models and of type `long` in the LP64 model, data structures manipulated in memory also must be 64-bit aligned. To avoid problems, be sure that existing data-packing code is "unpacked" when it manipulates data in memory.

For a detailed explanation of all these challenges and how you can address them, refer to Appendix 1 for access to documentation through your operating-system vendor.

GETTING STARTED

By now, you should have an initial understanding of the benefits of porting your application to the IA-64 architecture, the steps involved, and what you can do now to get started. Assessing the complexity of your application, especially in terms of third-party dependencies, is something you can start today. Concurrently, you can develop your porting plan, including selecting your target operating system and the best porting approach for that environment, enrolling your engineers in formal training, and acquiring the migration tools now available from the operating-system vendors. As soon as you complete your plan, you can begin modifying your code.

Not only can you start your IA-64 port now, but you *should* start your port now, because of the solid benefits to be gained. The earlier you identify third-party dependencies, the earlier you can help to ensure the vital libraries are ported. By porting now, you also can maximize engineering efforts that may already be underway toward other enhancements, such as a new 32-bit version of your product.

Most important, by porting now you can be ready to begin testing, optimization, and benchmarking the moment hardware is available—and to be among the first to deliver your customers the performance, scalability, and reliability advantages of the IA-64 architecture. For more information, go to the relevant sites listed in Appendix 1.

APPENDIX 1: TRAINING AND TOOLS

From Intel: To access an array of Intel porting documentation, including Web-based training materials, go to the following links:

- For an introduction to the IA-64 architecture as well as information on using the IA-64 instructions and cleaning code for the IA-64 architecture, go to http://developer.intel.com/vtune/cbts/ia64tuts/index.htm.
- For a technical summary of IA-64 architectural features go to http://developer.intel.com/design/ia64/downloads/adag.htm.
- For additional technical information, go to http://developer.intel.com/design/ia64/devinfo.htm.
- For an executive summary of IA-64 go to http://www.intel.com/eBusiness/enabling/engine.htm.

From the operating-system vendors: To access porting tools available from the major operating-system vendors, go to the following links:

- HPUX* 11.x: http://www.software.hp.com/products/IA64/index.html
- Linux: http://www.linuxia64.org/
- Modesto*: http://www.novell.com/whitepaper/iw/modesto.html
- Monterey: http://www.projectmonterey.com
- Win64*: http://msdn.microsoft.com/developer/news/feature/win64/64bitwin.htm

APPENDIX 2: DYNAMIC LIBRARY INTERACTION

It's a good idea to understand your options if your application depends on any third-party dynamic library that is not targeted for immediate IA-64 porting by its supplier. As explained earlier in this document, the library can remain unported if it will be accessed exclusively by IA-32 processes, but it should be ported if it will be accessed by any IA-64 process (32-bit or 64-bit).

If for some reason this is impossible, you must ensure that the IA-64 binaries accessing the library run in separate process spaces from those of the IA-32 binaries and that they use IPC to communicate with one another. This is necessary because all the major operating systems prohibit the mixing of IA-64 and IA-32 library instructions, even though the IA-64 architecture allows it.

To implement the necessary IPC, you can follow one of two approaches, as shown in Figure 2. You can (1) use surrogate binaries to manage the IPC translation with no changes to existing code or (2) create a wrapper for the library.

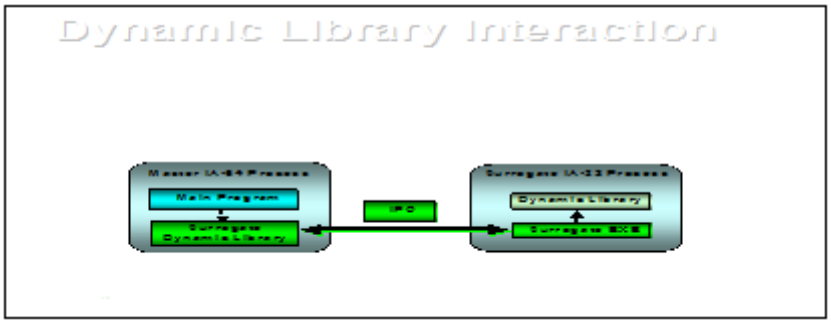

Figure 2: How to manage coexistence between an IA-32 dynamic library and IA-64 application code.