



# 64-bit Migration to Linux on Itanium®

## *Challenges, Advantages and Tools*

Soumitra Chatterjee  
Hewlett Packard



Gelato ICE: Itanium® Conference & Expo 2006

October 1 – 4 | Singapore

# Agenda

- Linux on Itanium®
  - The environment
- Migrating to Linux on Itanium®
  - Transitioning to 64-bits
- Common Pitfalls in 64-bit Migration
  - And how to avoid them
- Assessment and Migration Tools

# Linux on Itanium®

## *The Environment*



# Data Model

## Conventional Bit Size

Data Types	LP64	ILP32	LLP64
char	8	8	8
short	16	16	16
int	32	32	32
long	64	32	32
long long	64	64	64
pointer	64	32	64

# Data Model

- Linux on Itanium® follows the LP64 data model
  - long and pointers are 64-bit
  - int is 32-bit
- No native support for ILP32
- IA32 emulated support for x86
  - Intel Value Engine (IVE) on chip
  - Emulation layer in the kernel

# Structure Alignment

- On most systems, compilers align data types on a natural boundary
  - 32-bit data types are aligned to a 32-bit boundary
  - 64-bit data types are aligned to a 64-bit boundary
- In a structure, the compiler inserts filler (or padding) to enforce alignment
- Structure itself is aligned based on its widest member
  - On 64-bit systems, structures having 64-bit data type members will be aligned on 64-bit boundary

# Structure Alignment

	32-bit System		64-bit System	
struct align {	Size	Address	Size	Address
int a;	32	0x00	32	0x00
			32-bit padding	
double b;	64	0x04	64	0x08
int c;	32	0x0c	32	0x10
			32-bit padding	
long d;	32	0x10	64	0x18
};	Structure size 20 bytes		Structure size 32 bytes	

# Endian

- Endianism defines how bytes are addressed within integral and floating data types
- In little-endian implementations,
  - Least significant byte stored at lowest memory address
  - Most significant byte stored at highest memory address
- In big-endian implementations,
  - Most significant byte stored at lowest memory address
  - Least significant byte stored at highest memory address

# Endian

Byte order for a 64-bit long integer

Little endian	0	1	2	3	4	5	6	7
Big endian	7	6	5	4	3	2	1	0

long int value of 1025 examined as an array of bytes

Little Endian	Big Endian
<code>char*[0] = 0x01</code>	<code>char*[0] = 0x00</code>
<code>char*[1] = 0x04</code>	<code>char*[1] = 0x00</code>
<code>char*[2] = 0x00</code>	<code>char*[2] = 0x04</code>
<code>char*[3] = 0x00</code>	<code>char*[3] = 0x01</code>

# Linux on Itanium®

- 64-bit kernel
  - Always runs in IPF mode
- Little-endian native byte order
- LP64 Data Model
- No native ILP32 support
- IA32 emulated support for x86
  - Intel Value Engine (IVE) on chip
  - Emulation layer in the kernel

# Migrating to Linux on Itanium®



# 64-bit Advantages

- Virtual Memory
  - 64-bit applications can directly address 4 exabytes ( $2^{63}$ )
  - Itanium® provides a contiguous linear address space
- File System
  - 64-bit Linux allows file sizes up to 4 exabytes ( $2^{63}$ )
- Numerical Computations
  - Significantly faster due to hardware support
- Dates
  - Represented using 64-bit signed integer

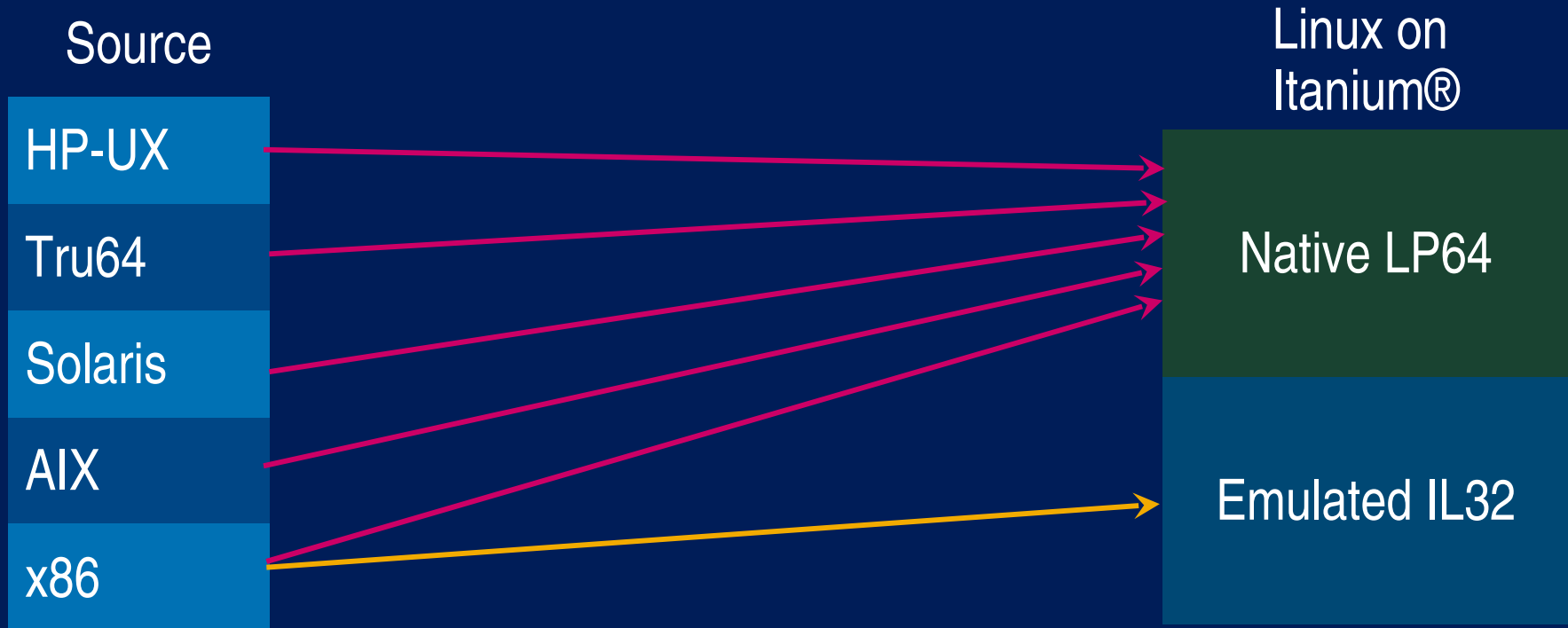
# When to Migrate?

- 64-bit benefits applications that
  - Manipulate very large data sets that are close to or must exceed the 4 GB (32-bit address space) limit
  - Are I/O bound and can use memory to perform disk I/O
  - Are compute-intensive (*e.g. database backend, CRM, ERP, eBusiness, crypto*)
  - Need to project dates into the future
    - Traditional Linux dates (32-bit signed) turns negative in 2038
- Applications that are already 64-bit

# Applications that benefit from 64-bit

- CAD / CAM applications
- Computer simulation applications
- Web-servers
- Multi-media applications
- High Performance Computing Cluster (HPCC) applications
- Database backend
- Business applications
  - Customer Relationship Management (CRM)
  - Enterprise Resource Planning (ERP)
  - eBusiness applications (such as online commerce stores)
- Financial applications that need to project dates into the future

# Migration Strategy



- Recompilation to native architecture
- Emulation using IL32 layer

# Preparing for 64-bit Migration

- Find all your source code
  - check under the couch!
- In-house libraries also need to be migrated
- Assembly code must be re-written
- Add warning options to Makefiles
  - Wp64 option for Intel compiler
  - Use cadvise / flexelint before build
- Acquire 64-bit versions of all 3<sup>rd</sup> Party libraries
- Use STK's and migration tools to point out issues

# Migration Methodology

- Identify dependencies
  - Third-party libraries, applications, compilers, tools
- Compile on IA-32 Linux if the migration is from a non-Linux platform
- Clean up the code
  - Fix compiler warnings
  - Use correct header files (e.g. API prototypes)
- Select the appropriate toolchain
- Build, test, optimize

# Common Pitfalls in 64-bit Migration

*And how to avoid them*



# Pointer/long Assignments

- On 32-bits systems, int and pointer/long are same size
  - Can be used interchangeably
- On 64-bits systems, truncation of higher 32-bits happens when
  - Assigning pointer/long to int
  - Casting pointer/long to int
- Implicit pointer/long return types are dangerous
  - e.g. calling functions that return pointer/long without prototypes

# Non-prototyped Functions

```
int main() {  
    char *p = malloc(20);  
    *p = 'A';  
}
```

- Although the above program works fine on 32-bit mode, may core-dump in 64-bit mode

# Non-prototyped Functions

- Will an explicit cast help?

```
void main() {  
    char *p = (char*)malloc(20);  
    *p = 'A';  
}
```

# Bit-shifting

- Data type of shift expression is the data type of the expression being shifted
  - Not the expression representing the shift amount
  - Not the target of the assignment

Expression	Type	Conversion	Result
<code>1 &lt;&lt; 31;</code>	32-bit int	Sign-extended	<code>0xffffffff80000000</code>
<code>1L &lt;&lt; 31;</code>	64-bit long		<code>0x80000000</code>
<code>1U &lt;&lt; 31;</code>	64-bit unsigned long		<code>0x80000000</code>

# Portable Bit Manipulation

- Setting all bits in a value
  - Using 0xFFFFFFFF will only work in a 32-bit system
  - Using 0xFFFFFFFFL will only set the low-order 32-bits on a 64-bit system
  - For a portable constant, use a signed long
    - long constant = -1L;
- Setting the most significant bit
  - Usually on 32-bits systems, 0x80000000 is used
  - For portability, use
    - 1L << ((sizeof(long)\*8) - 1);

# scanf() and printf()

- Be careful about conversion specifications
  - A %d conversion specification will truncate a 64-bit long to its least significant 32-bits
- In scanf(), misuse of conversion specification may result in incorrect values, segmentation fault or exception
- When a small integer is passed into printf(), it is widened to 64-bits
  - Sign extended, if appropriate
- Use standard macros for format specifiers
  - PRId32, SCId32

# Expressions and Assignments

Expression	Result	
	32-bit	64-bit
<code>int i = -2; unsigned k = 1;</code>		
<code>long n = i + k;</code>	-1	4294967295
<code>long n = (long)i + k;</code>	-1	-1
<code>long n = (int)(i + k);</code>	-1	-1

- unsigned int result not sign-extended during assignment
- i sign-extended to long, k promoted to unsigned long
  - 64-bit unsigned long result is not converted during assignment
- unsigned int result is cast to int, and sign-extended during assignment

# Performance Implications of 64-bit

- Applications may run slower in 64-bit mode than in 32-bit mode because the hardware contains structures (notably the TLB and Cache) that are fixed in size
- Packing these structures with 64-bit objects halves their capacity
- In 64-bit mode, long and pointers are 64-bit, resulting in a larger cache footprint

# Assessment and Migration Tools



# Compilers

- Intel C++ Compiler
  - Capable of specifically reporting 64-bit migration issues
  - Use option `-Wp64` to turn on 64-bit migration warnings
- GNU Compiler Collection
  - Includes compilers for C/C++ and other languages
  - Does not have any specific 64-bit migration warnings
  - Use option `-Wall,-ansi` and/or `-pedantic` to point out questionable constructs
  - Use option `-Wabi` (g++ only) to warn when the compiler generates code that is not vendor neutral

# Tools

- SPLINT
  - Tool for statically checking C programs
  - The option `–strict` is useful during porting (*quite verbose*)
- Programmer's Tool Kit
  - Set of tools for developing applications on Linux
- FLEXELINT
- HP Porting Kits
  - Solaris to Linux Porting Kit (SLPK)

# Takeaways



# Key Takeaways

- Making software 64-bit clean is very important
- Linux on Itanium® uses the LP64 little-endian standard
- Advantage for applications that manipulate large data sets/files
- Be aware of the 64-bit implications
- Use tools to aid in migration

# References

- HP Developer & Solution Partner Program  
<http://www.hp.com/dspp>
- Linux Programmer's Toolkit  
<http://www.hp.com/go/linuxtools>
- Software Transition Kits  
<http://www.hp.com/go/stk>
- SPLINT  
<http://www.splint.org/>
- FLEXELINT  
<http://www.gimpel.com/html/lintinfo.htm>
- HP Code Advisor  
<http://www.hp.com/go/cadvise>
- Intel Compilers  
<http://www.intel.com/cd/software/products/asm-na/eng/compilers/>

Questions?





i n v e n t