Porting Code to IA-64

David Prosser
Architect,
Development Systems
dfp@sco.com

Agenda

- Programming models
- Development environment and debugging
- Porting code to the different models
- Finding and fixing porting problems

IA-64 UNIX Programming Models

- IA-32 (Pentium® II processors, etc.)
  - as in UnixWare® 7 today
- ILP32
  - `int`s, `long`s, and pointers are 32 bits
  - new instruction set (IA-64 32 bit)
- LP64 (default)
  - `long`s, and pointers are 64 bits
  - new instruction set (IA-64 64 bit)
- No mixing permitted although supported by IA-64 architecture
  - one compilation model per process

Other IA-64 Programming Models

- ILP64
  - 64 bit `ints`, `longs`, and pointers
  - potentially fewer porting problems
  - no convenient 32 bit integer
- LLP64
  - 64 bit pointers; integers unchanged
  - model used by Microsoft NT
  - potentially breaks “portable” programs that mix pointers and integers
  - precludes 128 bit `long long`

Data Size and Alignment
(all have little-endian byte order)

C / C++ Data Types	ILP32 (IA-32)		LP64	
	Size (bytes)	Align. (bytes)	Size (bytes)	Align. (bytes)
char	1	1	1	1
short	2	2	2	2
int	4	4	4	4
long	4	4	8	8
long long	8	4	8	8
pointer	4	4	8	8
float	4	4	4	4
double	8	4	8	8
long double	12	4	16	16

IA-32 Environment

- **Binary compatible with UnixWare 7**
  - **supports the Intel published ABI**
- **Almost entirely handled in “user space”**
  - **thin layer between the kernel and your binary means minimal execution overhead**
  - **will take advantage of epc-based system calls**
- **Appropriate when single binary needed for IA-32 and Monterey IA-64 (or when there is no source)**

ILP32 (IA-64 32-bit) Environment

- IA-32 data layout compatible
- Performance similar to LP64
  - smaller data size (better cache use)
  - data conversion in/out of kernel
  - some misaligned data objects
- Fully supported–not just “intermediate step”
- Source compatibility
- Appropriate for recompile-and-go software

LP64 (IA-64 64-bit) Environment

- Highest performance
- UNIX industry-wide 64-bit model
- 64 bit “generic” ABI publicly available
  `http://www.sco.com/developer/gabi/contents.html`
- Processor specific ABI available from Intel
- All architecture’s features available
- Entire kernel built LP64
- Little-endian byte order
- Appropriate for new and high-end software

IA-64 Development and Debugging

- Single cc and CC compilation commands provide all compilation models
  - no mixing of models
  - supporting ELF tools work similarly
- Debugging provided for all models, with lowest levels matching the process
  - i.e., an IA-32 process sees %eax
  - but, an IA-64 process will see gp
- Controlled processes can have different models

IA-64 Compilation Defaults

- LP64
- Position independent code (PIC)
  - works best with IA-64
- System V dynamic linking
- Instructions and read/write data separated
- No inline assembly “escapes”
  - write complete assembly functions, but only when absolutely necessary

IA-64 Calling Convention

- Arguments are passed in 8 byte slots or multiples thereof
  - first 8 slots are in registers
  - high order bits unspecified for integer returns and arguments smaller than 8 bytes
- Special rules for passing and returning aggregates
  - especially for all-floating structures
- Function pointers do not point at code

Porting Code to ILP32 Model
Both IA-32 and IA-64 32 bit

Most IA-32 binaries just will work!

- **`/proc` file system will reflect the kernel**
  - **debuggers will need to be ported**
- **Exotic `ioctl`'s can be problematic**
- **System administrative files might change**

Porting Code to ILP32 Model
Only IA-64 32 bit

Lots of code will recompile and work!

- **"Machine specific" part of the user context differs from both UnixWare and AIX**
  - **more and different register sets**
- **Argument passing assumptions**
  - **aligned to 8 byte slots**
    - » **`long long` will not look like a pair of `longs`**
  - **extra alignment padding for `long doubles`**
  - **special aggregate handling**

Porting Code to LP64 Model

- Good code that also does not depend on byte order or external data formats will recompile and run correctly
  - generally, share/freeware code
  - uses prototypes and all appropriate headers
- Often old and stale code will work fine

HOWEVER

- Finding and fixing the problems that do happen is most of the rest of this talk

So, Why Port to LP64 Model?

- Need larger (64 bit) address space
- Need larger scalar arithmetic ranges
  - bigger basic data sizes (`time_t`, for example)
- Application Performance
  - IA-64 instruction set architecture
    - » faster than IA-32 instructions
  - no misaligned data
    - » alignment faults can be expensive

ILP32 ➡ LP64 Portability Issues

- Changes in relative integer sizes
  - `int` and `long`
- Changes in pointer/integer sizes
  - `int` and pointers
- Function calls without full declarations
- Objects changing size
- Stack layout changes
- System data types
- AIX 64 bit migration guide
  - `http://www.developer.ibm.com/`

64 Bit Enabled `lint`

- **Available at `http://www.sco.com/developer`**
  - **"64 bit UnixWare porting guide" also provided**
- **Supports ILP32 and LP64 models**
  - **`g64lint -K lp64` (default)**
  - **`g64lint -K ilp32`**
- **Complete set of header files and libraries**
- **Also, see `http://doc.sco.com`**

  `=> Software Development`
  `=> Programming in Standard C and C++`
  `=> Analyzing your code with lint`

Assignment Truncation of Integers

```
1   int int1, int2, int3;
2   long long1, long2, long3, retlong(int);
3
4   void f(void) {
5      int1 = long1;          /*64b => 32b*/
6      int2 = int2 * long2;   /*64b expr => 32b*/
7      int3 = retlong(long3); /*64b arg => 32b
8                               64b ret => 32b*/
9   }
```

```
assignment causes implicit narrowing conversion
    (5) int = long
    (6) int = long
    (7) int = long
```


Assignment Truncation of Integers

- Examine all narrowing assignments; correct as needed
- Use explicit casts where narrowing conversions are expected
  - unfortunately, this can then be a source for troubles later

```
5       int1 = (int)long1;
6       int2 = (int)(int2 * long2);
7       int3 = (int)retlong((int)long3);
```


Explicit Cast Improperly Applied

- **Apply narrowing casts to expressions**

```
1   int int1, r1, r2, r3;
2   long long1;
3
4   void f(void) {
5      r1 = long1 / int1;
6      r2 = (int)long1 / int1;   /*32b expr => 32b*/
7      r3 = (int)(long1 / int1); /*64b expr => 32b*/
8   }
```


Integer Pointer Conversions

```
1   int *pint1, *pint2;
2   long *plong1, *plong2;
3   void fint(int *), flong(long *);
4
5   void f(void) {
6      pint1 = (int *)plong1;
7      plong2 = (long *)pint2;
8      fint((int *)plong1);
9      flong((long *)pint1);
10  }
```

```
pointer cast may result in improper alignment
    (7)        (9)
```

- Use `-p` option to flag all pointer casts

```
pointer casts may be troublesome
    (6)        (7)        (8)        (9)
```


Integer Pointer Conversions

- **Examine all instances of incompatible pointer assignments**
  - **adjust size of objects based on range of values to be held in the object**
  - **use explicit casts to indicate intentional mismatch**
    - » **older memory management routines**
    - » **use `void *` for generic pointers**
    - » **`lint -p` will not flag `void *` uses**

Integer Expression Evaluations

- **Operands widened to “common type”**
  - **`int` – if operands are of type `int` or smaller**
  - **larger only if an operand is larger than `int`**

```
1   int int1, int2;
2   long long1;
3
4   void f(void) {
5      long1 = int1 * int2;         /*32b multiply*/
6      long1 = (long)(int1 * int2); /*32b multiply*/
7      long1 = (long)int1 * int2;   /*64b multiply*/
8      long1 = int1 * (long)int2;   /*64b multiply*/
9   }
```


Integer Expression Evaluations

- To get 64 bit results:
  - an operand of the expression must be either of type `long` or `unsigned long`
  - use wider constant or a cast if necessary
  - “widening” conversions percolate up the expression tree
    - » exceptions: shift operators and sequence points
- No assistance from `lint`

Integer Constants

- Type determined by shape and value
- Leading (and high order) zeroes only serve to denote octal – no other affect on size
- General rules:
  - decimal constants find first signed type that holds the value, small to large
  - other bases find first signed or unsigned type that holds the value, small to large
  - suffixes (combinations of `u` or `U`, and `l` or `L`, and `ll` or `LL`) generally restrict the choices

Integer Constants – Issues

- **Porting issues with code that:**
  - **does not take into consideration that integer constants may be more than 32 bits**
  - **assumes that `long` or `unsigned long` data is 32 bits**
  - **depends on specific behavior at an assumed data type length**

Integer Constants – Examples

- **Expression truncated at 32 bits**

```
long1 = long1 + 20000000 * 30000000;  /*32b expr*/
long2 = long2 + 20000000L * 30000000; /*64b expr*/
```

- **Expression depends on 32 bit truncation**

```
long1 += 0xffffffff; /*long1-1 for ILP32
                       long1+4294967295 for LP64*/
```


Integer Constants – Examples

- **Constant has `int` size, not "full size"**
  - **leading zeroes do not increase the size**

```
long1 &= ~0xffff0000;         /*clears 48 bits*/
long1 &= ~0x00000000ffff0000; /*clears 48 bits*/
long2 &= ~(long)0xffff0000;   /*clears 16 bits*/
long2 &= ~0xffff0000L;        /*clears 16 bits*/
```


Integer Constants – Examples

- **Shifts expecting 32 bit operands**
  - **can be hidden in macro expansions!**

```
ulong1 = (ulong1 << 5) >> 16; /*ILP32: keeps bits 11-26
                                 LP64: bits 11-58*/
long1 = (long1 << 5) >> 16; /*ILP32: might sign ext.11-26
                               LP64: bits 11-58*/

ulong1 = (ulong1 & 0x7fff800) >> 11;

long1 = (long1 << (CHAR_BIT * sizeof(long) – 27))
            >> (CHAR_BIT * sizeof(long) – 16);
```


Integer Constants – Guide

- Use of all constants should be reviewed
- Do not forget symbolic constants from `#define` directives
- Watch for:
  - 64 bit expressions where overflow or underflow may have occurred on a 32 bit sub-expression
  - octal or hex constants with 2^{31} as high order bit
  - expressions depending on truncation at 32 bits

Changing Pointer/Integer Sizes

- **Problem areas:**
  - **code that converts pointers to `int` or `unsigned int` with the expectation that pointer value is preserved**
  - **code that assumes pointers and `int`s are the same size in an arithmetic context**

Changing Pointer/Integer Sizes

```
1   int int1;
2   long long1;
3   char *charp;
4   void fint(int), flong(long);
5
6   void f(void) {
7      int1 = (int)charp;
8      fint((int)charp);
9      long1 = (long)charp;
10     flong((long)charp);
11  }
```

- `lint` flags conversions that can lose information

```
(7) warning: conversion of pointer loses bits
(8) warning: conversion of pointer loses bits
```


Changing Pointer/Integer Sizes

- **Pointer and `int` in arithmetic context**

```
1   #define BUSY 0x1
2
3   struct blk *blkp;
4
5   void f(void) {
6      /*...*/
7      blkp = (struct blk *)(BUSY | (int)blkp);
8      /*...*/
9   }

(7) warning: conversion of pointer loses bits
```


Changing Pointer/Integer Sizes

- All conversions of pointers from or to integers should be reviewed
- If necessary:
  - use `long` or `unsigned long`
  - use `intptr_t` or `uintptr_t` from `<sys/types.h>`

Lack of Prototyped Function Declaration In Scope

- *default argument promotions*
  - integer promotions for parameters smaller than `int`
  - undefined behavior if called function expects a larger type
    - » ILP32 and LP64 compilation models
  - IA-64 calling convention
    - » padding bits are unspecified

Lack of Function Declaration In Scope

- Implicit return type of `int`
  - caller will sign-extend the presumed 32 bit `int` value value if used with a 64 bit type
  - if a pointer or `long` actually returned, the high order bits are lost
  - even more interesting if structure actually being returned

Lack of Prototyped Function Declaration In Scope

- Use `lint` on all source files that make up a binary to find:
  - implicitly declared functions (point of call)
  - functions declarations with “old-style” parameter lists (point of call)
  - functions with an implicit `int` return type
  - argument types used inconsistently
  - function return types used or declared inconsistently

Objects Changing Size

- Object whose sizes will differ
  - pointers, `long` and `long double`
- Object whose sizes might differ
  - `double`, `long long`
  - alignment differences may effect padding
- Only issue if data is shared between an ILP32 binary and an LP64 binary

Objects Changing Size

- Developer responsibility to define matching data objects in each model
- If necessary, use `#ifdef`'s
  - `#if LONG_MAX > 0x7fffffff`
    - » defined in `<limits.h>`
  - use "model" predicate to control definition

```
#if #model(ilp32)
#if #model(lp64)
```


Fixed Size Data Types

- **Defined in <sys/types.h>**

Fixed Size Data Types		ILP32 (IA-32)		LP64	
signed	unsigned	Size (bits)	Align. (bytes)	Size (bits)	Align. (bytes)
int8_t	uint8_t	8	1	8	1
int16_t	uint16_t	16	2	16	2
int32_t	uint32_t	32	4	32	4
int64_t	uint64_t	64	4	64	8

  - **64 bit size still has alignment differences**

Predefined System Type Changes

- Types intimately bound to address space size are either `unsigned long` or `long`
- Certain values such as wide characters and file mode bits are adequately represented in 32 bits

Predefined System Type Changes

UNIX System Type	UnixWare 7		Future Releases		
	C Data Type	Size (bytes)	C Data Type	ILP32 Size (bytes)	LP64 Size (bytes)
mode_t	unsigned long	4	unsigned int	4	4
ptrdiff_t	int	4	long	4	8
size_t	unsigned int	4	unsigned long	4	8
ssize_t	int	4	long	4	8
wchar_t	long	4	int	4	4
wint_t	long	4	int	4	4
wuchar_t	unsigned long	4	unsigned int	4	4

Summary

- You can “have it your way”, using the model that meets your needs
- Porting to either ILP32 model is easy
- Porting to LP64 may well require some code analysis and changes
- Use `g64lint` as your first analysis step
- Testing/certification costs will dominate, no matter which model used

Downloading `g64lint`

- `http://www.sco.com/developer`
  - 64-bit Tools and Technical Information
    - 64-bit UnixWare Porting Guide
    - `g64lint` tool
    - 64-bit driver porting information
- Questions or Comments
  - `unison64@sco.com`
  - `chibib@us.ibm.com`
- Porting guide from 32 bit AIX to both Monterey ILP32 and LP64 coming soon