64-Bit Programming Models: Why LP64?

Participation from: Digital Equipment Corporation, Hewlett-Packard Company, IBM Corporation, Intel Corporation, Novell Inc., NCR Corporation (formally AT&T GIS), Santa Cruz Operation Inc., Sunsoft Inc., and X/Open Company Ltd. (a.k.a., Aspen Data Model Committee)

EXECUTIVE SUMMARY

The Open Systems community is at an optimum moment to make a choice regarding how 64-bit architectures will be supported by our implementations. The technical arguments included in this paper defend the position that **LP64** is a better solution for 64-bit programming than other potential models (**ILP64 & LLP64**) . The key evaluation issues are portability, interoperability with 32-bit environments, standards conformance, performance effects and transition costs; as we analyzed each of the data models against these evaluation metrics, **LP64** was our clear choice.

PROBLEM STATEMENT

Demand for a larger flat address space is the key opportunity for 64-bit systems. There is much discussion today in the computer industry about the barrier presented by 32-bit addresses. 32-bit addresses to byte granularity are capable of selecting one of 4 gigabytes (GB) of memory. Disk storage has been improving in areal density at the rate of 70% compounded annually, and drives of 4 GB are readily available with higher-density drives coming shortly. Memory prices have not dropped as sharply, but 16 MB chips are readily available with 64 MB chips in active development. CPU processing power continues to increase by about 50% every 18 months, providing the power to process ever larger quantities of data. This conjunction of technological forces along with the continued demand for systems capable of supporting larger data bases, larger simulation models and newer data types (e.g., full-motion video) have generated requirements for support of larger addressing structures in todays systems.

Major chip architectures have begun to provide direct addressing of uniform address spaces larger than 4GB. As these chips and systems based on them come to market, there is an opportunity to agree on the programming model

that will be used with them, thus avoiding unnecessary fragmentation. This is clearly best done within the environment of the Open Systems specifications as implementations exist for all the architectures. Further, if the various implementers can agree on some of the details, then application developers will be faced with a single environment from multiple vendors, easing on-going support costs.

TECHNICAL CHOICES

The C programming language lacks a mechanism for adding new fundamental datatypes. Providing 64-bit addressing and scalar arithmetic capabilities to the C application developer involves changing the bindings or mappings of the existing datatypes or adding new datatypes to the language.

There are three basic models that can be chosen, **LP64**, **ILP64** and **LLP64** as shown in the following table. The notation describes the width assigned to the basic data types. **LP64** (also known as 4/8/8) denotes ***long*** and ***pointer*** as 64 bit types, **ILP64** (also known as 8/8/8) means ***int***, ***long*** and ***pointer*** are 64 bit types and **LLP64** (also known as 4/4/8) adds a new type (***long long***) and ***pointer*** as 64 bit types. Most of today's 32 bit systems are **ILP32** (that is, ***int***, ***long*** and ***pointers*** are all 32-bits wide). The majority of C Language programs written today for Microsoft Windows

3.1 are written for the Win-16 APIs which is an **LP32** (***int*** is 16 bits, while ***long*** and ***pointers*** are 32-bits) model. The C definitions on the Apple Macintosh are also **LP32**.

Datatype	**LP64**	**ILP64**	**LLP64**	**ILP32**	**LP32**
char	8	8	8	8	8
short	16	16	16	16	16
_int32		32			
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
pointer	64	64	64	32	32

C language standards specify a set of relationships between the various data types but deliberately do not define actual sizes. Ignoring the non-standard types for a moment, all three 64-bit pointer models satisfy the rules as specified.

A change in the width of one or more of the C datatypes affects programs in obvious and not so obvious ways. There are two basic sets of issues: (1) data objects defined with one of the 64-bit datatypes will be different in size from those declared in an identical way on a 16 or 32-bit system, and (2) assumptions (not those specified within the C standard, but used anyway by the developers of particular pieces of code) about the relationships between the fundamental datatypes may no longer be valid. Programs depending on

those relationships often cease to work properly.

LLP64 preserves the relationship between ***int*** and ***long*** by leaving both as 32-bit datatypes. Objects not containing ***pointers*** will be the same size as on a 32-bit system. Support for 64-bit scalar data is provided by adding a new, non-portable scalar datatype such as ***__int64*** or ***long long***. **LLP64** is really a 32-bit model with 64-bit addresses. Most of the runtime problems associated with the assumptions between the sizes of the datatypes are related to the assumption that a ***pointer*** will fit in an ***int***. To solve this class of problems ***int*** or ***long*** variables are changed to ***long long***, a non-standard datatype. This solution is optimized for the first class of problem and is dependent on the introduction of a new datatype.

An **LLP64** operating system is forced to either change the datatype definition of many of the system programming interfaces (API's) or introduce a new set of 64-bit wide interfaces. Most of these interfaces have been stable for almost 25 years across all versions of the UNIX operating system. Thus, **LLP64** requires extensive modifications to existing specifications to support those places which should naturally become 64-bit wide.

ILP64 attempts to maintain the relationship between ***int, long*** and ***pointer*** by making all three the same size (as is the case in **ILP32**). Assignment of ***pointers*** to ***int*** or ***long*** variables does not result in data loss. On the other hand this model either ignores the portability of data or depends on the addition of a 32-bit datatype such as ***int32*** or ***__int32.*** This is a potential conflict with existing typedefs, and is especially contrary to the spirit of C development, which has avoided embedding size descriptions into fundamental datatypes. Programs that must preserve the size and alignment of data are forced to use the non-standard datatype and may not be portable.

The world is currently dominated by 32-bit computers, a situation that is likely to exist for the foreseeable future. These computers run 16 or 32-bit programs, or some mixture of the two. Meanwhile, 64-bit CPUs will run 32-bit code, 64-bit code, or mixtures of the two (and perhaps even some 16-bit code). 64-bit applications and systems must integrate smoothly into this environment. Key issues facing the industry are the interchange of data between 64 and 32-bit systems (in some cases on the same system) and the cost of maintaining software in both environments. Such interchange is especially needed for large application suites (like database systems), where one may want to distribute most of the programs as 32-bit binaries that run across a large installed base, but be able to choose 64-bits for a few crucial programs, like server processes. **ILP64** implies frequent source code changes and requires the use of non-standard datatypes to enable interoperability and maintain binary compatibility for existing data.

LP64 takes the middle road. 8, 16 and 32-bit scalar types (***char***, ***short*** and ***int***) are provided to declare objects that maintain size and alignment on 32-bit systems. A 64-bit type (***long***) is provided to support the full arithmetic capabilities and is available to use in conjunction with pointer arithmetic. Programs that assign addresses to scalar objects need to specify the object as ***long*** instead of ***int***. Programs that have been made 64-bit safe can be

recompiled and run on 32-bit systems without change. The datatypes are ***natural***, each scalar type is larger than the preceding type.

As a language design issue, the purpose of having ***long*** in the language anticipates cases where there is an integral type longer than ***int.*** The fact that ***int*** and ***long*** represent different width datatypes is a natural and common sense approach and is the standard in the PC world where ***int*** is 16 and ***long*** is 32-bits.

EVALUATION CRITERIA

The programming model choice described in the last section can be made individually by each of the system vendors, or jointly through an implementers agreement amongst multiple vendors. We argue that the Open Systems community, most particularly the application developers, are best served if there is a single choice widespread in the emerging 64-bit systems. This removes a source of subtle errors in porting to a 64-bit environment, and encourages more rapid exploitation of the technology options. Also, this is an opportune moment to make such an agreement, since the early shippers (Digital and SGI) have already selected the same model (namely, **LP64**), while other vendors have not yet committed their shipping products to a choice.

The remainder of this paper describes the evaluation criteria we suggest using to make a selection for the industry, and assesses the **LP64** and **ILP64** models against these criteria. The **LLP64** model is not, in our view, a satisfactory basis for widespread adoption and use since it requires extensive modification to existing standards.

PORTABILITY

A major test for any model is the ability to support the large existing code base within the Open Systems arena. The investment in code, experience, and data surrounding these applications is the largest determiner of the rate at which new technology is adopted and spread. In addition, it must be easy for an application developer to build code which can be used in both existing and new environments.

At this point, there is experience at Digital and SGI with the realities of porting applications to an **LP64** based 64-bit programming environment. The Digital UNIX product requires that ALL applications run in an **LP64** runtime environment. The SGI product provides a mixed environment supporting **ILP32** and **LP64** models, but many applications currently shipping on the SGI systems are already using the **LP64** model.

The Digital and SGI experiences prove complementary facts. Digitals shows that it is possible for ISVs and customers to port large quantities of code to a 64-bit environment, while producing and inter-operating with 32-bit ports elsewhere. SGIs experience shows that code can be improved to be compilable for either 32 or 64-bits and still be able to interchange data in the

more tightly-coupled fashion expected by processes on the same system.

Although we are beginning to see a number of applications grow to the point of requiring the larger virtual address space there hasnt been a requirement for a 64-bit ***int*** data type. The majority of todays 64-bit applications previously ran only on 32-bit systems (usually some flavor of UNIX), and had no expectation of a greater range for the ***int*** data type. In such cases, the extra 32 bits of data in a 64-bit integer are wasted. Future application requiring a larger scalar data type can use *long.*

Other language implementations will continue to support a 32-bit integer type. For example, the FORTRAN-77 standard requires that the type **INTEGER** be the same size as **REAL**, which is half the size of **DOUBLE PRECISION**. This, together with customer expectations, means that FORTRAN-77 implementations will generally have **INTEGER** as a 32-bit type, even on 64-bit hardware. A significant number of applications use C and FORTRAN together -- either calling each other or sharing files. Such applications have been amongst the quickest to find reason to move to 64-bit environments. Experience has shown that it is usually easier to modify the data sizes/types on the C side than the FORTRAN side of such applications, and we expect that such applications would require a 32-bit integer data type in C regardless of the size of the ***int*** datatype.

Nearly all applications moving from a 32-bit system require some minor modifications to handle 64-bit pointers, especially where assumptions about the relative size of ***int*** and ***pointer*** data types were made. We have also noticed assumptions about the relative sizes ***of int***, ***char***, ***short*** and ***float*** datatypes that do not cause problems in an **LP64** model (since the sizes of those datatypes are identical to those on a 32-bit system) but do in a **ILP64** model. Our experience suggests that neither an **LP64** nor an **ILP64** model provides a painless porting path from a 32-bit system, but that all other things being equal, smaller datatypes enable better application performance

INTEROPERABILITY WITH 32-BIT SYSTEMS AND OTHER 64-BIT SYSTEMS

For many years, we expect the bulk of systems shipping in the computer industry will be based on 32-bit programming models. A crucial investment for end-users is the existing data built up over decades in their computer systems. Any solution must make it easy to utilize such data on a continuing basis.

Unfortunately, the **ILP64** model does not provide a natural way to describe 32-bit data types, and must resort to non-portable constructs such as ***__int32*** to describe such types. This is likely to cause practical problems in producing code which can run on both 32 and 64 bit platforms without **#ifdef** constructions. It has been possible to port large quantities of code to **LP64** models without the need to make such changes, while maintaining the investment made in data sets, even in cases where the typing information was not made externally visible by the application.

Most ***int***s in existing programs can remain as 32 bits in a 64-bit environment; only a small number are expected to be the same size as ***pointers*** or ***longs***. Under **LP64** very few instances need to be changed.

With **ILP64**, most ***int***s need to change to ***__int32***. However, ***__int32*** does NOT behave like 32 bit ***int.*** Instead ***__int32*** is like ***short*** in that all operations have to be converted to ***int*** (64-bits, sign extended) and performed in 64-bit arithmetic.

So, ***__int32*** in **ILP64** is not the same as ***int*** in **ILP32**, nor the same as ***int*** in **LP64.** These differences will predictably cause subtle hard-to-find bugs.

STANDARDS

The Open Systems community is technically driven by a set of API agreements embodied in specifications from groups such as X/Open, IEEE, ANSI, ISO and OMG. These documents have developed over many years to codify existing practice and define agreement on new capabilities. Thus, the specifications collectively are a major value to the system developers, application developers and end-users. There is a body of work on verifying that implementations correctly embody the details of the specification and certify that fact to various consumers. These verification suites are also part of the "glue" that keeps us a community. Any 64-bit programming model cannot invalidate large quantities of these specifications (with their extensive detailed descriptions) and expect to achieve wide adoption.

Currently shipping, **LP64** based operating system have met and passed many of the existing specifications and verification suites. It is our observation that there was no major philosophical barrier in doing so, but rather much detailed review of critical items buried within the specifications and verification suites. As a community, we know by demonstration that **LP64** systems can comply with the commercially important standards; there is NO such demonstration today for **LLP64** or **ILP64** systems

In particular, most standards, but particularly the language standards such as the ANSI C specification deliberately are silent in enforcing width decisions for basic data types since history has shown an assortment of choices made to reflect underlying architectures. This leads to deliberate ambiguity in the meaning of certain code samples, both as they move between different C compilers and when they move between optimization levels on specific C compilers. Some of these can occasionally cause practical problems, but well-written code and tools such as lint have eased this problem significantly. Moreover, the experience of porting applications between various vendor platforms has significantly helped find problems like this. Nonetheless, such examples exist both as demonstration points and as practical problems - they require significant intellectual energy to get them right. With **LP64**, we have the accumulated experience of several implementers and application developers to help.

PERFORMANCE CHARACTERISTICS

We have argued that technology advances at economic levels enable 64-bit computing to become widespread. However, for some time the acceptable economic boundaries for many problems will be a barrier. Thus, crucial to rapid adoption is attention to inherent performance differences between the models. We see two categories of differences: (1) instruction cycle costs to properly implement the defined model, and (2) memory system costs (at all levels of the memory hierarchy) to transport to the places required.

Instruction cycle penalties will be incurred whenever additional cycles are required to properly implement the semantics of the intended programming model. However, compiler writers have extensive experience with the type of optimizations that are available.

For example, in **LP64** its only necessary to perform sign extension on ***int*** when you have a mixed expression including ***long***; most integral expressions do not include ***longs***. Compilers can be smart enough to only sign extend when necessary. Many current architectures (Alpha, MIPS, Sparc V9 and PowerPC) dont have a problem because there is a 32-bit load that performs sign extension, although they may have the analogous problem with 32-bit unsigned ***int***. Given that most CPUs will spend much of their time executing 32-bit operations (whether running 64-bit programs, or simply doing 32-bit operations in 64-bit code), it seems difficult to understand why many implementations would penalize 32-bit operations.

In our porting experience, these "inner loop" issues have NEVER become major performance determiners for commercial applications, and the current balancing act between memory cycle times and CPU cycle times leaves open issue slots which can often absorb the additional cycles.

A much larger practical effect in some commercially important applications comes from the consumption of additional memory and the costs of transporting that memory throughout the system. 64-bit integers require twice as much space as 32-bit integers. Further, the latency penalty can be enormous, especially to disk, where it can exceed 1,000,000 CPU cycles (3 *nsec* to 3 *msec*). ***int*** is by far the most frequent data type to be found (statically and sometimes dynamically) within C and C++ programs.

Some software vendors have experimented with an **ILP64** model, which can be approximated on **LP64** systems by changing all ***int*** declarations to ***long.*** In these cases, the conclusion reached after these experiments was not to use **ILP64**, since the application did not benefit from the additional range of ***int*** values and did not wish to pay the performance penalty of extra memory use.

TRANSITION FROM CURRENT INDUSTRY PRACTICE

By far, the largest body of existing code already modified for 64-bit

environments runs on **LP64** based platforms. The vendors of these platforms have worked with most of the application developers whose support is critical to any widespread adoption of 64-bits in the community. The practical problems of such ports have been resolved in a fashion that is demonstrably successful based on years of market experience.

There have been a few examples of **ILP64** systems that have shipped (Cray and ETA come to mind). These attempts have not had the broad market base that **LP64** systems have had, although they do demonstrate that it is feasible to complete the implementation of an **ILP64** environment.

Applications already modified for **LP64** will need additional effort in code audit, changes to usage of ***int***, and performance tuning to run on **ILP64** systems. Many large ISVs would need separate code pools to support this difference. It is hard to see why this change provides them any value to compensate for the additional effort.

Applications not yet modified for any 64-bit systems have the experience of others to guide them - experience expressed as tools to identify troublesome constructs, and as porting documents. These experiences become the practical guidance needed to both encourage adoption and avoid pitfalls.

SUMMARY

Each of the evaluation criteria are subject to extensive further exploration (a quote comes from Jim Gray - Computing is fractal; wherever you look, there is infinite complexity). We have argued that each issue supports a choice of **LP64**.

- **Portability**, especially for combined FORTRAN and C code, is enhanced with **LP64** and the most common types of problems that occur are susceptible to automatic detection.
- **Interoperability** is improved by the ability to use a standard data type to declare data structures that can be used in both 32 and 64-bit environments.
- **Standards conformance** has been demonstrated both in the practical sense of porting many programs and in the formal sense of compliance with important industry standards.
- **Performance** has been a major and effective selling theme for both **LP64** systems, and the memory size penalty for unneeded 64-bit integers can be very high for some applications.
- **Transition** from the current industry practice is smooth and direct following a path grooved with experience and demonstrated success.

All this, as well as ***natural*** use of the native C datatypes to support all the widths needed in a 64-bit system make a compelling argument for the inherent advantage of **LP64.**

Read other technical papers.

Read or *download* the complete Single UNIX Specification from http://www.UNIX-systems.org/go/unix.