nant, rather than merely important, we might also reach an ILP64 conclusion. Were we to be simply defining a new architecture of 64 bits, and not an extension to the existing SPARC family, we might also look more favorably upon ILP64.

However, the very fact that we are gathered as part of an organization chartered for product interoperability, rather than as a subcommittee of the ANSI-C effort, illustrates the dominant importance to our organizations of both maintaining and increasing the level of interoperability between all members of the SPARC family. To make a choice that omits the 32-bit data type from our standard for binary interoperability, the most widely used data type in our current software base, cannot be the right answer. To require an immense effort at penetrating the 32-bit market with a *new* data type at this point in its lifetime also cannot be the right answer.

If we truly value interoperability and the time of our software developers, we will not present yet another issue for them to “cope with”, and instead we will make the choice that yields maximum return for them — that choice, and the choice of Sun Microsystems, is LLP64.

We have previously expressed skepticism over the utility of the second assumption, given the value we place upon interoperability with respect to portability. By definition, such programs can not be portable *and* interoperable while simultaneously exploiting the properties of a larger native size — for they will either scale a problem beyond "the network" that must deal with it, or else they will simply fail. It's extremely rare to see a portable, interoperable program that is independent of any size knowledge, and we confess that we're unsure that there exists a real-world example. However, despite our skepticism, we'll evaluate both LLP64 and ILP64 from the perspective of either assumption.

5.4.1 Code "knows" Data Size

LLP64 renders the porting of this code trivial by maintaining the assumption "as is." Although it is possible that there will remain portability issues surrounding pointer arithmetic, it seems unlikely that these would fail without other portions of the program similarly failing. **lint** or similar tools can automate the detection and correction of such errors should they occur.

ILP64 makes porting code in this class very difficult. The basic assumptions upon which the code and algorithms are based are changed, and there is little one can do to automate the detection of errors so introduced.

5.4.2 Code assumes "natural sizes"

LLP64 complicates the porting of this code by denying the code the ability to automatically obtain the benefits of the 64-bit architecture. It may introduce errors due to assumptions based upon the ability to perform pointer arithmetic and the size of the result of such arithmetic. Mitigating this factor is the prospect that **lint** or similar tools can automate an investigation for such issues.

ILP64 presents the "natural" scenario for this code — the port is trivial and automatically obtains the benefits of the 64-bit architecture.

5.5 SPARC ANSI-C Conformance

LLP64 extends ANSI-C by the addition of the type **long long**. This extension does not invalidate conforming programs. We maintain our assumption that this extension will take place regardless of whether or not SPARC-64 exists — many vendors are already using or plan to provide it. ANSI-C will ultimately come to recognize it. LLP64 is otherwise ANSI-C conforming.

ILP64, if it provides no 32-bit type, is ANSI-C conforming. However, no one believes that SPARC-64 can exist without a 32-bit type, which will be the most prevalent type used in interoperability scenarios. Thus, to actually work, ILP64 requires a *change* to ANSI-C to introduce either size-specific types (**__int*n***, etc.) or a new, mid-range, natural type (**medium**). The success of either type is predicated upon its penetration into the 32-bit world, a world that has already chosen to invest in **long long**, and thus unlikely to accept a new change.

6. Summary and Recommendation

The results of our comparisons are summarized in Table 2. Given our goals and assumptions, LLP64 is the clearly superior choice.

Goal (Section)	LLP64	ILP64
32 ⇔ 64 (5.1)	Yes	No
SPARC ⇔ Not (5.2)	Yes	No
Shared Source (5.3)	No **#ifdef**	**#ifdef**
Port: Sizes (5.4.1)	Easy	Hard
Port: Natural (5.4.2)	Modest	Easy
ANSI-C (5.5)	Extension	Change

Table 1: Evaluation Summary

The cost of choosing ILP64 is to incur interoperability difficulties and to expend our most precious resource solely for the sake of what is a primarily academic argument. Our competitors have made it clear that they do not intend to make this mistake[8] — we should not be lulled into believing that we can do so either.

That the conclusions reached in our position differ from positions reached in others is due less to technical considerations than they are to value judgements over what is important. The ILP64 arguments have their greatest strengths when viewed from a perspective of portability as the dominating concern and when considering a 64-bit architecture in isolation. Were we to consider portability domi-

8. Mashey, John: USENET group comp.arch on 24-Mar-92, Message: <kstnndINNss2@spim.mips.com>. With tongue in cheek: "TO ALL COMPETITORS: when you do 64-bit machines, ..., under *no* circumstances should you offer an option to let **sizeof(long)** = 32. ... *Ignore* foolish, idiotic software developers... *Make* them do the right thing, even if it means your port takes 18 months longer than your competition."

5. Evaluation vs. Goals

LLP64 and ILP64 are evaluated with respect to their suitability at meeting the goals described earlier. We omit an evaluation of IP64 because it would not be different from that LLP64 except with respect to ANSI-C conformance.

5.1 Interoperability with SPARC-32

LLP64 is completely interoperable with the SPARC-32 data types, making readily accessible to native SPARC-64 programs the data and protocols of the SPARC-32 world. Save data structures containing pointers, the databases of one are completely interchangeable with those of the other.

ILP64 is nearly completely non-interoperable with the SPARC-32 data types. It will require complete reworking of source code to contain either abstract types or different algorithms and library functions delineated with **#ifdefs** or machine-dependent sources in order to restore it. ILP64 turns what was to be a zero software investment into a major project for the writers of software. Examples of where this work will be required are:

- all file system formats and utilities that operate upon them (including those database vendors that construct "private" formats upon raw media);
- file formats containing binary data such as those produced by Frame and Interleaf;
- communications products such as the X11 window system, all **rpcgen** or equivalent source files and in general all sources using Sun RPC.

Even after this work is finished, interoperability is not provided in any transparent fashion. Assuming that, for instance, Frame intended for their binary formats to be "ANSI-C portable" rather than SPARC interoperable — then users would have to use the "interchange format" between two members of the SPARC family.

5.2 Interoperability with non-SPARC

For big-endian 32-bit architectures, or for the exchange of data using "presentation layer" technologies such as XDR, LLP64 is as interoperable with non-SPARC architectures as it is with SPARC-32. ILP64, for these cases, presents all the same problems — although since expectations for exchange are lower across family boundaries the pain of these considerations might be lessened.

ILP64 is at a significant disadvantage even if it is assumed that data exchange always happens through a "presentation layer." The extant sources that use these layers know both the sizes of the presentation transport and the sizes of the data being marshalled for presentation. This means that, although it is possible for an ILP64 to utilize a "network **long**", it will only be possible through use of a not-yet-defined 32-bit type, whose use is predicated upon source modification. The "native" **long** will require the use of the network "**hyper**" (the network 64-bit type), which of course would be an incompatible change to whatever protocol is involved.

ILP64 again takes an advantage in leverage that could have accrued to the SPARC-64 community and reduces it to one that will require effort to achieve.

5.3 Shared Source

LLP64, by preserving the data types and with its assumption of **long long** in 32-bit software, preserves both data structure and algorithm compatibilities. It will not be necessary to isolate support for **long long** file offsets behind either **#ifdef** guards or machine-dependent source modules. Evolution of the software will affect SPARC-64, SPARC-32, and most non-SPARC architectures simultaneously. "Interchange" or "translation" technologies will not be required in order to use 32- and 64-bit implementations of a single software package on a given piece of data.

LLP64 does require that those who wish 64 bit data types must explicitly request them, even on a 64-bit architecture.

ILP64 without **long long** makes it impossible to share source between SPARC-32 and SPARC-64 without considerable effort in isolating both data structure declarations and algorithms using known 64-bit sized interfaces in SPARC-32 code. ILP64 with **long long** reduces this effort, but still requires a significant reworking of the extant data structure declarations.

5.4 Portability

The impact upon portability of each choice is primarily a function of which of two assumptions you wish to make about the nature of the code to be ported:

1. The code "knows" how big an **int** and/or **long** and/or **long long** are; or

2. The code is written to rely not upon any sizing constraints save those in inherent in the C language — it will run on machines with 57-bit **long**s and 35-bit **int**s.

C Type	PC Macintosh	SPARC-32	SPARC-64 LLP64	SPARC-64 IP64	SPARC-64 ILP64
char	8	8	8	8	8
short	16	16	16	16	16
“medium”	—	—	—	—	32
int	16 and 32	32	32	64	64
long	32	32	32	32	64
“long long”	64	64	64	64	—
pointer	32	32	64	64	64

Table 1: Data Type sizes

The first two columns agree in all respects except for the size of **int**: historical practice has accepted variation here, and there are many existing programs that port between SPARC-32 and the PC-class systems: the latter being the origin for many such programs.

4.1 SPARC-64 Common Ground

In all three SPARC-64 columns, the sizes of **char**, **short**, and **pointer** agree. That no one has seriously suggested any assignment for the first two of these save for what they have always been suggests to us an appreciation of the value of not challenging assumptions that have never been violated. That no one has suggested any assignment for **pointer** other than 64 bits also suggests an appreciation of one of the few unchallenged assumptions in C code.

4.2 LLP64

LLP64 makes choices consistent with the view that the use of types has generally been made with knowledge of their sizes. It preserves the fact that **long** has effectively never been changed. Communications protocols, media, and files written with either of the machines in the first two columns is easily achieved. Porting of software that uses **long long** is similarly easy.

LLP64 obligates those who really do want a 64-bit type to say so. And, it assumes an extension to ANSI C in the form of the **long long** data type an extension that will not affect conforming programs.

4.3 IP64

IP64 builds upon LLP64 in that it preserves long-standing assumptions but adds the feature of having a “natural” 64-bit type. Since the industry already copes with varying sizes of **int**, this is not a new burden for the industry.

The principal negative with IP64 is that it violates the ANSI-C rule that an **int** is no larger than a **long** — an assumption that has never yet been violated. And, the fact that **int** changes from 32 bits will probably increase the cost of this option to the writers of software compared to LLP64.

4.4 ILP64

ILP64 makes choices consistent with the view that programs are written without regard to the value of data type sizes, but are instead based upon the structural relationships between them. Its only commonality with SPARC-32 is the sizes of **char** and **short.** It violates the never-changed assumption over the size of **long**. Interoperability through communications protocols, media, and files written in 32-bit environments can only be achieved through large changes to the source base.

Some claims for ILP64 state that it obviates the need for a **long long** type — thus, portability between SPARC-32 and SPARC-64 will be achieved only through large amounts of conditional compilation for code involving the use of **long long** on SPARC-32 and other 32-bit architectures.

Within the confines of ANSI-C, the only way in which interoperability with 32-bit software can be achieved through an extensive change to source code to use bitfields. Alternatively, a new “natural type” must be added to the language (e.g., **medium**) or an extensive list of “size explicit” integral types. Both changes would have a significant source impact.

tween SPARC-32 and SPARC-64 machines be equally transparent. A user running a desktop publishing application should not discover that a publication can be operated upon by one or the other, but not both SPARC architectures.

Achieving this goal obtains the maximum leverage from the critical software-writer resource — as it requires no work from them and creates no pressures to do additional work through barriers to interchange.

3.2 Interoperability with non-SPARC

The preceding goal dealt with the most important consideration of product interoperability between members of a family. However, in the market reality that renders SPARC the underdog with respect to PC-class computers, it is important to ensure that interoperability is maintained throughout the environment in which SPARC machines will operate. This means using the protocols and data formats as they will exist "native on the wire." A product can be successful in the computing environment of the 1990's only to the extent that it can be successfully added to and interoperate with an extant system — and this means be *useful* in that system through ready operation upon and exchange of data that already exists.

3.3 Preserve Resources: Shared Source

Most software that exists in "native" form on SPARC-64 will be built from source for which other machines are also targets. These machines will be 32-bit architectures, hopefully including SPARC-32. Having obtained a SPARC-64 port, it will be necessary to maintain that port as the software evolves — and the cost of such maintenance can be seen to vary monotonically with the amount of "cases" that embed themselves in the source code.[7] By "cases" we mean both variances that manifest themselves as distinct portions of the source code hierarchy, and in-line differences that appear as preprocessor managed conditional compilation ("**#ifdef**").

3.4 Portability: Lower Barriers

Most software that will exist on SPARC-64 will not only target other architectures, but will also originate there. As the architecture that will only rarely be "first choice" as a target, and even more rarely the "flagship" target, SPARC-64's success will be heavily influenced by being a "cheap win" that maximizes the software writer's gain while minimizing their up-front costs. In the long-term, of course, maintenance and evolution costs will dominate, thus the lower priority of this goal with respect to the previous one.

To achieve this goal requires meeting the state of the software as it is — not as we might wish it to be. Here it becomes important to understand what the real assumptions are as well as the costs incurred by violating them. Given the previously identified shortcomings in the language, it is clear that it will be rare to not violate *some* assumption — it behooves us to find solutions that minimize the costs that then occur.

3.5 SPARC-64 ANSI-C Conformance

We already know that the ANSI-C standard is deficient with respect to our needs. Still, deficient does not mean "invalid" — and as a tool to help achieve the preceding goal, conformance with ANSI-C will be a boon. However, it is less of a goal than the primarily economic-based practica identified in the other goals. It is important to recognize that standards exist to codify an economic consideration of some kind: of money, people, time, or some combination. Thus, the view of whether a standard is succeeding is very much dependent upon whose economy is being maximized. As we have previously described, the economics of interoperability dominate those of portability: and ANSI-C, as a standard supporting portability economics, will be similarly dominated.

4. The Choices

Table 1 summarizes the dominant choices in the industry: for the PC space, SPARC-32 (and other 32 bit architectures), and the three operative proposals for SPARC-64: LLP64, IP64, and ILP64. We have omitted other "possible" choices because discussion to date has already eliminated them from discussion. We have similarly omitted floating point choices from the table as these are also non-contentious.

Given the environment into which SPARC-64 machines will be placed, the outcome of our choice is to construct a table that will contain three of these columns: the first two (describing the unalterable state of the environment) and one of the following three.

7. "Vary monotonically" means only that costs increase when the number of "cases" also increase — but we make no claims as to the proportions of the variance and merely recognize that there exist cases that are less than linear, and others that vary as the square or cube.

change in terms of future editions of the SCD, there will be first-class usage of **long long** in system interfaces well before there are any 64-bit systems. This effort is not restricted to SPARC systems — competitors such as MIPS already have such a type on their architectures, and Microsoft is expected to embed such a type in system interfaces offered with their NT system software product.

An examination of the ANSI-C standard evokes the realization that it is not prepared to deal with 64-bit issues. Although we were able to survive with only three sizes for integer types for years, as described above many compilers have already chosen to extend the language with more types. Fortunately, most compilers have used similar designs, and there is consensus as to how to do this in the 32-bit architectures.

However, the use of ANSI-C in defining other standards has introduced problems. More precisely, a particular "style" of ANSI-C use has caused the problem. In a number of system interfaces, the type **int** has been used with the assumption that it would be the "right" size for the architecture. However, this can lead to imprecision when different C compilers for the same architecture make different choices.[5] For example, in SPARC-32, **ptrdiff_t** is defined to be an **int**[6]. It is easy to argue that **ptrdiff_t** should have been defined as a **long**, which is what it is on a Macintosh. A Macintosh has the same size for **pointer** as SPARC-32, but uses a 16-bit **int**. It is simply coincidence that **long** and **int** are the same size on SPARC-32. This suggests that assumptions based upon empirical observations of the state of various standards will be at least occasionally suspect.

The arrival of **long long** will make such assumptions much more than suspect. It will threaten a number of system interfaces: in particular, those defined to return "the largest possible integer" — clearly, when files can be greater than 32-bits in length, operations defined to operate upon or return only a 32-bit quantity can not suffice. Thus, similar to what was done to the **seek()** call (which became **lseek()**), we must be careful to avoid an assumption that an architecture change occurs simultaneously with an interface change — and similarly not assume that an interface change will be obviated by a predicted architecture change. The corollary negative assumption for SPARC-64 environments is that simply "floating" interfaces to their "natural sizes" in a 64-bit environment will not occur, due to its threat to the body of 32-bit software that will already exist. Extending the interface, instead of simply redefining it for SPARC-64, allows SPARC-32 programs to access the objects that SPARC-64 machines will be creating on the network.

5. This is not hypothetical: the 80386 C compiler must make different choices for **int** based on whether it is compiling for the Windows 3.0 vs. Windows-32 libraries.
6. *System V Application Binary Interface, SPARC Processor Supplement*, pg 6-48.

2.5 Derived Assumptions

From the preceding observations on the arrival of **long long**, we can derive a number of other assumptions that, although suggested by the previous paragraphs, are not stated explicitly. These are:

- SPARC-64 programs always use 64-bit pointers;
- SPARC-32 programs will not change except to access new functionality;
- SPARC-32 programs will sometimes manipulate 64-bit quantities; and
- SPARC-64 programs will often manipulate 32-bit quantities.

3. Goals for SPARC-64 Data Types

The remainder of this section describes an ordered list of goals to be achieved in the choice of C data types for extending the SPARC architecture to 64 bits. These goals derive from the success factors and technical and market considerations discussed previously. We recognize that as *goals*, they are things we must strive to achieve — but they are not *constraints* which must be satisfied perfectly.

3.1 SPARC-32 "Emulation" & Interoperability

SPARC-32 programs must execute efficiently on SPARC-64 machines. By "efficiently" we refer not only to execution performance, but also with respect to the facility with which such programs interact with each other and with native SPARC-64 programs. In particular, it is important that neither developers nor users of SPARC-64 systems develop different "vocabularies" and therefore usage characteristics that imply knowledge of which programs operate in emulation vs. those that operate native. It is "failure" in this context to have a SPARC-32 program fail to operate for any reason other than the lack of a 64-bit pointer. It is "success" if native and emulation versions of the same program can operate upon the same data.

Similarly, it is important that exchanges be-

tunate if we were to take backward steps in evolving the SPARC architecture.

However, arguments involving machine types invariably omit the most dominant machine of all — the aggregate called "the network" — and in this respect the network may really *be the* computer. The network's data types are defined (in our space) by the presentation services offered by XDR. This is evolving rapidly, as evidenced by recent actions in the Object Management Group (OMG). The OMG, seeking technology to allow software developed in different programming languages on different systems to communicate, endorsed an approach used by a number of systems: an Interface Definition Language (IDL). IDL unambiguously specifies the range of values that are to be communicated, and embeds these ranges in a type system similar to that of C. The sizes chosen are the same as those now in evidence on SPARC-32 and other 32-bit architectures.

Thus, we believe that not only has historical precedent established and permitted assumptions over size to take hold in the industry, we also believe that this is a trend that will become *more* emphasized over time, rather than lessened. This trend is supported by the economics driving the dominance of interoperability over portability, and thus will occur without regard for historical ruminations over language design.

2.3 Assumptions and the Test of Time

Earlier discussions and exchanges have produced a catalog of assumptions that C programs, and in particular, C programs for 32-bit architectures, might have assumed.[4] We use this catalog to determine what assumptions have stood the test of time, and therefore more likely to be either implicitly or explicitly used as fundamental in constructing programs.

2.3.1 Assumptions that are violated in 32-bit code

The following are assumptions that it is claimed that 32-bit code can make that in fact are violated in exchanges between 32-bit systems. This table assumes the existence of **long long**, an assumption whose validity is pursued later in this text.

- **int**=32 bits: 80x86 and Apple Macintosh computers have a 16-bit **int.**
- **int**=**long**: 80x86 and Apple Macintosh computers have a 16-bit **int** and a 32-bit **long**.
- **int**=**ptr**: 80x86 and Apple Macintosh computers have a 16-bit **int** and a 32-bit **pointer**.
- **int**=register size: Apple Macintosh computers have 16-bit **int** and 32-bit registers. 80x86 can be argued to have either 16- or -32-bit registers.
- **int**>=all: Apple Macintosh and 80x86 computers have both **long** and **pointer** types larger than **int**. SPARC-32 and MIPS have 64-bit **long long**s.
- **long**>=all: SPARC-32 and MIPS have 64-bit **long long**s and 32-bit **long**s.
- **ptr**>=all. SPARC-32 and MIPS have 64-bit **long long**s and 32-bit pointers.

2.3.2 Assumptions violated, but not in 32-bits

The following are assumptions that, while generally safe in 32-bit code, have not always been true, having been violated at least once. While perhaps a stronger basis for "important" assumptions than the previous section, nonetheless they are less valid than assumptions that have never been violated.

- **ptr**=32 bits: PDP-11's had 16-bit pointers.
- **long**=**ptr**: PDP-11's had 16-bit pointers and 32-bit **long**s.
- **long**=register size: PDP-11's had 32-bit **long**s and 16-bit registers.

2.3.3 Assumptions *never* violated

The following are the remaining assumptions from the catalog: there is no existence proof of their ever having been violated in the community of general-purpose software. It thus appears more risky to break them than any of the other assumptions, if habits reinforced by time are to be used as any guide.

- **long**=32 bits.
- **ptr**=register size.

2.4 Emergence of a 64-bit type in SPARC-32

The type system is evolving even without the advent of true 64-bit architectures. Many vendors of 32-bit systems have already, or will shortly, add data types that support access for 64-bit objects. Our own efforts in the SPARC International "big SIG" to define new operations and library routines to operate upon objects that exceed the natural size of the machine are evidence of this. Assuming that this effort will yield both product and standards

4. Sundarrajan, Prabakar, "Rationale for 64-bit C Type Sizes", HaL Computer Systems, Inc., March 9th, 1992.

tions. This group will expect SPARC-64 to be just like SPARC-32, and expect to change few things in order to support the new functionality. They will expect no change whatsoever to gain interoperability. Changing the definition of sizes will require them to engage in work proportional to the differences.

It can be argued that these changes can be accomplished through rote means. Indeed, simply editing all header files and replacing instances of **long** with a **typedef**'ed abstract type might suffice. However, this change is a massively intrusive one — and to a software developer whose time is precious, it is a questionable use of their energies. That computers can be used wreak havoc over their sources will not be a comfort to the testing organizations — who will have to start from scratch with not only all configurations of the software but also all active versions. Unless one represents the "flagship" architecture for a software vendor, such expenditures are made as "luxury events" — time and material permitting.

2.1.2 Those who don't know

People who write software in this class expect each architecture to be correct in its own right. Their assumptions about data type sizes are not about their value, but instead upon the relationships of those values. They wrote and tested only to the portability guarantees in the standard, and their code will work just as well on a 32-bit machine as on a machine with a 57-bit **long**, 35-bit **int**, and 27-bit **short**. Not changing the sizes of these data types to take advantage of new architectures would require them to edit their programs to use types that did, or else foreswear such advantages.

A problem with the code of "those who don't know" is that of external representation. At a source level, only **char** is used in external interfaces and in data shared with other programs. However, even at this level the program must be making an assumption over *one* data type size or else forego interoperability altogether. A **char** on a PDP-10, a Honeywell-6000, or UNIVAC-1100 is going to be 9 bits — and thus using it as a conveyance for data observed by a machine with 8-bit **char**s is problematical without other agreements. Such programs achieve interoperability only by restricting their manner of interchange to codesets with a predefined and restricted set of values such as ASCII.

2.1.3 Who to believe?

In the absence of the ability to read the minds of all programmers, or even all of their code, we must make a decision over whose assumption we're going to believe ourselves. We can at best make educated guesses through an examination of the code available to us, of the assumptions that history has permitted people to make, and of our own experience.

2.2 20 Years of Assumptions

C is old, having existed for almost half of the era of von Neumann computers and for well over half of most uses of computing. As a result, we have a considerable history from which to draw data about "what people assumed" based on "what *could* they have assumed?"

For most data types, there is no controversy over whether or not the size ought to be defined and what value that size must be[3]. **char** will be 8 bits, **short** 16 bits, **float** 32 bits, and **double** 64 bits. These values are rarely questioned and even more rarely changed, and especially so in the volume market from which we expect to obtain most of our software. Even the size of **long**, the target of most of our controversy, has been 32 bits in practically every C compiler ever made. Pointers are assumed to be dependent upon the size of the architecture, and are not usually stored in files or communicated in protocols and thus not an impediment to interoperability. Although the size of **int** is still arguable, most people acknowledge that it will be different for different architectures, and it is almost always defined to be the same as **short** or **long**, making it redundant. Many of the extant uses of these latter types are in fact an attempt to *fix* the property that **int** will vary — and thus in the case of **long**, the attribute that it will be at least 32-bits in length has come to mean that it ***is*** 32 bits in length — at least for those who know.

In fact, for most of the last 20 years, for practically every architecture, there has been agreement over the sizes of **char**, **short**, **long**, **float**, and **double**, and there has been understanding that **int** and **pointer**s vary with the architecture. Thus, you never see the latter two used in data structures used to support interoperation. With standardization of character sets and floating point formats, the industry is within striking distance of easy and reliable data exchange methods. It would indeed be unfor-

3. We know that there have been and are computers that have made other choices. However, these systems occupy niche markets which will not be the basis of general-purpose computing in the 1990's. It is for the latter that we do ABIs.

portability one. The importance of this distinction is easily illustrated in the economics of the software business. The work to port software is expended a few times for a particular software product by a few people. Many copies of the resulting binary are then distributed. The cost of portability (or lack of it) is amortized across all the copies that are distributed. However, the value of interoperability is accrued by the many users of the software over and over again throughout its life.

Achieving interoperability is a complex problem — but it eventually comes down to two crucial agreements over:

- inter-component communication; and
- shared data accessed by different components.

Both of these agreements require parties to have equivalent definitions for the information being transmitted or stored. Various techniques have been developed to cope with these problems. For example, database systems use data definition languages, RPC systems use packet definition languages and marshalling routines, protocol hierarchies generalize these into "presentation layers." In each case, the goal is for different parties to use the same specification of the format of the information, eliminating the possibility of multiple inconsistent specifications. Once there is agreement upon the specification, a variety of mechanisms can be used to communicate or access the data.

1.4 SPARC-64 success factors

The environmental considerations discussed thus far collectively describe necessary constraints that must be satisfied for SPARC-64 to become successful. Summarizing, these are:

1. Economics make interoperability the dominant concern for purveyors of software;
2. Interoperability is fundamentally based in a shared specification over data and its access methods;
3. These economics define and constrain the activities of the most critical resource in making a architecture a success — the software writer; and
4. The conditions that create these economics and at which SPARC is at a current disadvantage are likely to persist.

Put succinctly, these simply state that those who write the software must perceive that a SPARC-64 system is interoperable with the rest of environment, *and especially so with SPARC-32 systems*. A solution that leaves SPARC-64 and SPARC-32 interoperability at the same level as SPARC-64 and other architectures deletes any advantage we might have accumulated from building a 64-bit SPARC as opposed to a 64-bit architecture with no discernible heritage.

2. Factors and Assumptions on C Data Types

The extensive considerations of the choice of sizes for SPARC-64 is the result of, and an indication of, the fact that the choice of sizes has an important affect on the software base. The process of choice introduces many questions: what assumptions have been made in existing code, and what issues were not anticipated when the code was written. We have collectively recognized that our standards and existing practice are imperfect in the face of the demands presented by a 64-bit architecture, although we have different opinions over the "right" way to address these imperfections.

To resolve these differences, we need to establish a common context within which to make a choice. The environmental considerations for SPARC-64 systems are one part of that context. With respect to the choice of data type sizes the remainder of the context can be obtained through the assumptions over current choices and their evolution independent of SPARC-64.

2.1 There are two kinds of programmers...

...those who know the size of data types, and those who claim they do not.

2.1.1 Those who know

The first group has, either wittingly or not, embedded in their programs choices in data type usage that wed their program to attributes of their target's data type assignment. It has been academically popular to sneer at such programs as the result of bad practice. In some cases, it is. In other cases, the program had to operate in an environment in which homogeneity was not an option — and often these programs had to *know* size attributes (such as: **long** is 32 bits, **short** is 16 bits) in order to operate at all. These attributes are used in external interfaces and data shared with other programs. The enshrinement of these choices in the SPARC ABI enhances the view that these things are of known, permanent, sizes.

Regardless of whether or not the reasons are technically justifiable, this group has written and tested their programs with the current size defini-

architecture “family.”

This level of heterogeneity will not be short-lived. Unlike the transitions from 16- to 32-bit architectures that occurred over the last decade, in which most 32-bit architectures entirely displaced the corresponding 16-bit architectures in a short time, the 32-bit to 64-bit transition will happen more slowly. We expect the coexistence of SPARC-32 and -64 machines to continue into the next century, as well as coexistence with other 32-bit architectures, especially IBM PC-compatible and Apple Macintosh computers.

In fact, not only do we expect them to coexist, we expect that SPARC-32 machines will outnumber SPARC-64 machines for most of the lifetime of the SPARC-64 architecture. And we further expect other architectures to dominate the overall computer market for the foreseeable future. As a result, most of the software that will run native on SPARC-64 machines will also (and probably first) run on SPARC-32 and other 32-bit architectures. This condition will probably hold for all software that does not absolutely require a 64-bit architecture, which includes most current and envisioned applications, the operating system, and other platform software.

Because the SPARC-64 instruction set is a superset of the SPARC-32 instruction set, it is possible for “SPARC-32 emulation mode” to be implemented using software techniques that allow SPARC-32 programs to run at practically the same speed as the native SPARC-64 applications.[1] It is likely that many applications will not need to be “ported” to SPARC-64 in order to be used effectively. This advantage will help the market penetration of SPARC-64 bit machines, as the considerations of distribution and support costs as well as system administration issues will make transparent use of SPARC-32 executables important. The model for success in this space is exemplified by the Intel 80x86 transitions, as opposed to the PDP-11 to VAX transition experienced by DEC.

1.2 ISV’s

The ideal to which most software developers aspire is to have a large amount of “architecture-spanning” software with a small amount of additional “architecture-specific” software. This software base is then used to produce different executable files. The now pervasive ability to share files over networks supports this model, often making it as easy to develop for multiple architectures as simply saying “**make**” on each target.

Despite this technological change, a software developer will tend to prefer a particular architecture (generally the one with greatest volume for their product.) It is common for the product to be shipped first on that architecture, and the edition of the product on that architecture is typically the “flagship” edition. As a result, purveyors of “lower priority” architectures need to mitigate the differences between their architecture and the “flagship” ones. And, in the race to marketplace success, it’s important to recognize that the most critical resource to conserve is the time and energy of the software writer — not just the energy required to perform an initial “port”, but on a continuing basis.

It is certainly true that there will be software that is designed specifically for the SPARC-64 architecture. There will also be software vendors who will consider it their primary architecture. Those developers who are motivated to target SPARC-64, and those customers who value its added capabilities will be willing to devote some effort to fully exploit it. However, an important reason for being part of the SPARC architecture family is to have available on the platform a larger body of software than might be attracted otherwise. To convince those developers who are primarily focused upon other architectures (*especially* SPARC-32) to contribute to SPARC-64, we must make it as easy as possible for them to support it and, having done so, make it as easy as possible for the users of the software to use it sensibly in the context of the total system environment.

1.3 Interoperability dominates Portability

In an earlier era, software portability was crucial issue. Most customers of software had access to the source and expected to carry it to different architectures. Architectures were numerous and changed frequently.

But in the modern computing environment, and in the market of the 1990’s, interoperability between programs is more important than portability of software.[2] The existence of SPARC International and the fact of our current considerations demonstrate this: SI is an interoperability body — not a

1. Existence proof: SunOS 4.x binary compatibility in SunOS 5.x. using the precursor technique to what SunSoft plans to do with the extended SPARC architecture.

2. This is not to say that portability is *un*important, simply that it is false economy to sacrifice interoperability for portability.

Extending SPARC Data Types to 64 Bits

Michael L. Powell & Robert A. Gingell

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View CA 94043

ABSTRACT

In extending the SPARC™ architecture to 64 bits, careful design will allow us to make an asset of our SPARC-32 investments, support convenient and efficient interoperation among the SPARC family, and allow programs to access the additional power and functionality of SPARC-64 machines. The design decisions we make must consider widely-used non-SPARC architectures, since these architectures will either be the source or common target of much of our software.

We examine the environment in which we are making this decision, our perspectives on the issues to be addressed and the goals to be satisfied, and examine the leading alternatives in light of these. We recommend the proposal that best satisfies these goals and assumptions, and comment upon the ramifications of it.

1. The SPARC-64 Environment

As we design the software view of the SPARC-64 architecture, we must consider the environment in which we expect to find SPARC-64 machines. The "environment" in this case includes the physical, technical relationship between SPARC-64 and other machines, and also the dynamics and economics of the software business for both suppliers of systems as well as those software developers upon whom our product success is predicated.

1.1 Configurations

SPARC-64 machines will almost always be connected to networks containing SPARC-32 machines. Moreover, it will be common for SPARC-32 binaries to be running on SPARC-64 machines. In both cases, whether by sharing files over a network or through the access of "native" files in "compatibility mode", it will be commonplace for the same source code and run-time data to be available to both 64- and 32-bit SPARC instruction sets. As illustrated by Figure 1, a decade of networking software development has made transparent sharing of data between machines a reality. It is typical today for users to be unaware of the architecture of the machine upon which the data is actually stored. It is

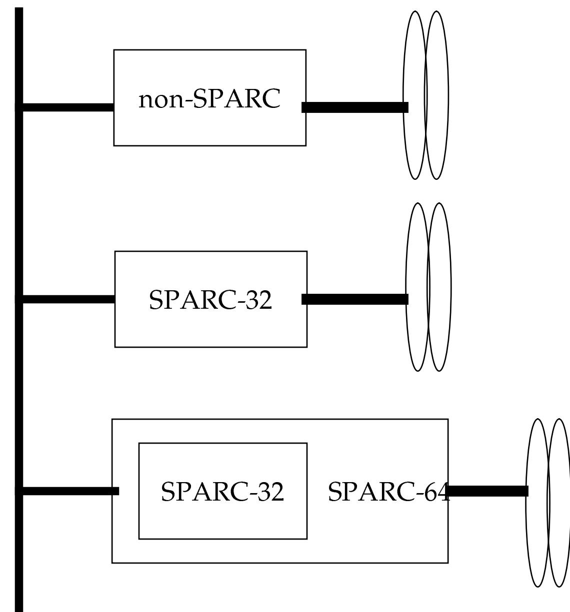

Figure 1: File Access Configuration

becoming increasingly typical that users are unconcerned about the architectures that process that data, and it is a "feature" of our SPARC-64 designs that users should not have such concerns with respect to SPARC-32. This is, after all, the point of an