XCL : An Xlib Compatibility Layer For XCB

Jamey Sharp Bart Massey
Computer Science Department
Portland State University
Portland, Oregon USA 97207–0751
{jamey,bart}@cs.pdx.edu

Abstract

The X Window System has provided the standard graphical user interface for UNIX systems for more than 15 years. One result is a large installed base of X applications written in C and C++. In almost all cases, these programs rely on the Xlib library to manage their interactions with the X server. The reference implementation of Xlib is as old as X itself, and has been freely available in source form since its inception: it currently is a part of the XFree86 [xfr] distribution.

Unfortunately, Xlib suffers from a number of implementation issues that have made it unsuitable for some classes of application. Most notably, Xlib is a large body of code. This is of most significance on small platforms such as hand-held computers, where permanent and temporary storage are both limited, but can also have performance disadvantages on any modern architecture due to factors such as cache size. In addition, because of Xlib's monolithic nature, it is difficult to maintain.

The authors' prior work on the X protocol C Binding (XCB) is intended to provide a high-quality but incompatible replacement for Xlib. While XCB is believed to be suitable for most new application and toolkit construction, it is desirable to support the large installed base of legacy code and experience by augmenting XCB with an Xlib-compatible API.

This desire has led to the construction of a new library, the Xlib Compatibility Layer (XCL), that is binary-compatible with frequently-used portions of Xlib while being significantly smaller and easier to maintain. Benefits are demonstrated for both existing and new applications written for Xlib. In particular, the significant share of existing knowledge and written material about Xlib remains applicable to XCL. Also, XCL can significantly ease the migration path from Xlib to XCB.

1 The X Window System

The X Window System [SG86] is the *de facto* standard technology for UNIX applications wishing to provide a graphical user interface. The power and success of the X model is due in no small measure to its separation of hardware control from application logic with a stable, published client-server network protocol. In this model, the hardware controller is considered the server, and individual applications and other components of a complete desktop environment are clients.

Development of X began in 1984, and it has become a mature and stable specification that many vendors have implemented for their particular hardware and operating environments. There is now a huge installed base of client applications: X is available for most modern computer systems, and is typically the default on UNIX systems.

To date, most client software for X has been built on top of one or more libraries that hide various details of the protocol, as illustrated in figure 1. Many applications are built using a GUI toolkit, such at Xt [AS90], Qt [Dal01], or GTK+ [Pen99]. These toolkits themselves, however, are almost invariably built on top of Xlib [SGFR92], a library that provides C and C++ language bindings for the X Window System protocol. It is also not uncommon to build applications directly atop Xlib.

2 Xlib and XCB

The authors' recent work has included development of a new X protocol C binding, XCB [MS01], that is intended as a replacement for Xlib in new applications and toolkits. XCB has a number of interesting features, but the XCB API is quite different from the Xlib API: XCB is not intended as a plug-compatible replacement for Xlib in existing applications.

XCB and Xlib are both designed to be C library interfaces to the X Window System network protocol for X

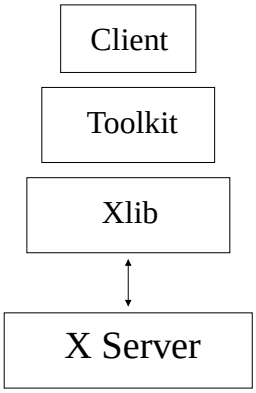

Figure 1: Xlib's role in the X Window System

client applications. However, XCB exchanges Xlib's numerous features for smaller size and enhanced performance. XCB is a lower level API than Xlib: much of the built-in functionality of Xlib (such as caching, display property management, and internationalization) is expected to instead be implemented as separate libraries above XCB, with XCB handling only interaction with the X server at the protocol level.

Some of the Xlib features omitted from XCB are important for getting reasonable performance or proper behavior from an X application. Nonetheless, we believe that these features do not belong in the core protocol library, and should instead be built on top of XCB.

Some of the differences between Xlib and XCB are worthy of detailed consideration.

2.1 Code Generation

The reference implementation of Xlib's core X protocol support alone consists of around 400 files comprising about 100,000 lines of hand-written code. Though the X protocol has a well designed mechanism for protocol extension, and the XFree86 X server allows for reasonably straightforward server-side implementation of these extensions, Xlib has not made the client-side task particularly easy.

In contrast, XCB provides a domain-specific language for specification of the binary encoding of the core protocol and of the majority of extensions. Automated tools translate these protocol descriptions into C implementations. The domain-specific language has significant advantages for maintenance, as well as for implementation of new features and extensions in XCB.

Protocol descriptions may be easily verified against the published specifications for the core X protocol and extensions. Experience with XCB has shown that a brief inspection of the XCB protocol description for a broken request can quickly lead to a correct fix. This benefit is due to the language making the structure of a request clear, while hiding implementation details in a single location in the source: the protocol-to-C translator.

This encapsulation has also made possible the implementation of a number of useful features, including request marshaling (described next) and tracing of requests and events. Without a common nexus for implementation of these features, the effort required to implement them would have been prohibitive.

2.2 Request Marshaling

For requests that pass to the server a list of independent items, such as a collection of arbitrary line segments, Xlib provides a form of transparent batching known as request marshaling. Marshaling allows client applications to draw individual lines, for example, from different sections of code in rapid succession, without incurring overhead from many similar request headers being sent to the server with small payloads.

Most requests are unsuitable for marshaling, for any of several reasons. If the request is expected to return a reply, combining requests would cause too few replies to be generated. Many requests send only a fixed-length data section with their request header: in these cases, there is generally no room to place additional data. In some cases, requests are not idempotent, so that combining requests would result in different behavior than sending the requests individually. `SetClipRectangles` is one example of this case. (In this paper, we will refer to X protocol requests by the name given to them in the X protocol manual. We will refer to Xlib or XCB functions by their function name: for example, `XSetClipRectangles`.)

XCB also supports marshaling, but treats it as a request attribute that may be specified in the description of the binary encoding of the X protocol. Because of this design, marshaling in XCB is generically supported anywhere in the core protocol or in any extension, although it should only be used if the conditions for correctness given above are satisfied. In the reference implementation of Xlib, marshaling is only used in the places where it is most likely to help: for instance, it will marshal multiple calls to `XDrawLine` but not calls to `XDrawLines`. While this is a wise optimization of programmer time when each function is hand-coded, XCB's use of automated code generation techniques allows it to be more thorough.

2.3 Latency Hiding and Caching

XCB has an easy-to-use latency hiding mechanism intended to allow client applications to get useful work done while waiting for information to return from the server. In XCB, this is accomplished by returning only a placeholder for an expected reply to a server request. Acquiring the placeholder takes no more time than would a request that does not expect a reply. The application can then convert this placeholder into the actual reply data at any time. When the actual data is requested the requesting thread may or may not block. Obtaining the reply data represented by the placeholder takes a small amount of time in most cases: it is quite likely that a request's reply data will already be available by the time the application asks for it.

The reference implementation of Xlib has a mechanism for the same purpose, called an async-handler, based on a callback model where one registered function after another is called until one of the callbacks claims responsibility for the data. However, use of Xlib's mechanism is considerably more complex.

Xlib also does extensive client-side caching. Implementation of various caches built on top of XCB is planned as future work, but XCB does no caching. The assumption is that caching can be better managed by a higher layer, for example a toolkit (such as Xt) or a convenience library. In some instances, caching is actually undesirable, either because memory is scarce or because an application needs to know what requests it is generating.

2.4 Thread Safety

Following the successful application of threads in Java's Swing GUI framework, XCB has been designed from the start with the needs of multi-threaded applications in mind, while still supporting single-threaded client applications. While Xlib was designed to support threaded applications, and while that support is not unusable, there are known race conditions that cannot be eliminated without changing the Xlib interface. In particular, if an application needs the protocol sequence number of the request it is making, Xlib forces it to make the request and, without any lock held, query the last sequence number sent. If another thread issues a request between these two events, the first thread will obtain the wrong sequence number. XCB simply returns the sequence number from every request.

2.5 Interfaces

Xlib and XCL have different function signatures for each request in the protocol. In some cases, such as Xlib's `XCreateSimpleWindow`, the differences are substantial. Still, in many cases these are simple reorderings of the same parameters, since both libraries provide APIs for the same X protocol. The ordering of fields in the protocol specification is optimized primarily for packing efficiency. Xlib uses a parameter ordering based on the parameter ordering of the reference library from version 10 of the X specification. Since X10 compatibility is no longer an important issue, XCB attempts to re-use the existing X11 protocol documentation by keeping its parameter order identical to the order of the fields in the binary protocol specification.

2.6 Feature Comparison

Thanks to the modular design of XCB, applications need only link against the small quantity of code that they actually need. In contrast, Xlib is a monolithic library supporting a variety of disjoint feature sets. Some features are not directly related to the core protocol:

- A color management system consists of mechanisms that enable colors to displayed consistently across a variety of hardware. Within Xlib is Xcms [SGFR92, Ber95], a complex set of functions for manipulating colors that includes support for device-independent color-spaces. This feature, though a good idea, is layered atop the RGB-based device-dependent color descriptions of the X protocol. While this functionality was included in Xlib in anticipation of its widespread adoption, it has in fact been little-used by applications, and does not appear in XCB. There exist stand-alone color-management libraries [lcm, Gil99] that do not depend on X at all. New code probably should use one of these libraries for color management.

- Xlib has extensive support for internationalization. In particular, it has functions for working with strings of 16-bit characters (such as `XwcDrawText`) and for interaction with input methods. This functionality is not intrinsic to the X Window System, but is conveniently separable. Only the internationalization support inherent in the core protocol is included in XCB. Xlib's internationalization support is becoming less relevant to modern X applications as libraries such as Xft [Pac01b] replace the core protocol font support in order to add anti-aliased fonts and other features unavailable in Xlib.

- Xlib has functions for performing many different kinds of searches on its event queue, including search for particular types of events and on particular windows. XCB provides only a generic queue

traversal, allowing layers closer to the application to provide implementations of predicates that select events.

- Xlib has convenience functions that build on the X protocol's notion of window properties to provide a *resource database*, a sophisticated configuration mechanism for X applications. Since the idea of a resource database is not inherent in the protocol, it is not supported at all by XCB, though the property primitives it relies on are. It would be straightforward to build a compatible resource database atop XCB using window properties.

3 XCL: Xlib Compatibility Layer

XCB has significant advantages over Xlib in some environments, especially small platforms such as handheld computers. However, there are more than 15 years' worth of applications and toolkits built using Xlib. For most existing software, the benefits obtainable by using XCB are outweighed by the effort required to port to it. As an alternative, the XCB design allows it to be used as a replacement "lower layer" for Xlib, efficiently handling X protocol interactions for existing software.

The Xlib Compatibility Layer (XCL) library is an attempt to provide an Xlib-compatible API atop XCB. While Xlib does attempt to layer portions of its implementation, the division between upper and lower layers in Xlib is only visible upon careful examination of the library. Identifying this split is key to providing an Xlib-compatible C library implementing the most commonly used portions of Xlib as a layer on top of XCB, as illustrated in figure 2.

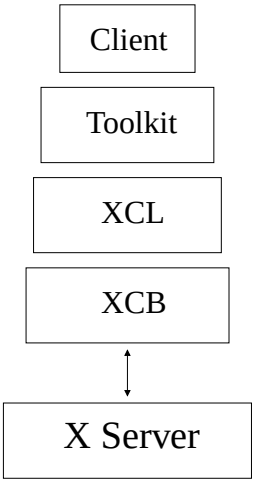

Figure 2: XCL's role in the X Window System

3.1 Xlib Coverage

XCL provides a significant fraction of the functionality of Xlib. This includes adapters from Xlib's protocol request functions to XCB's, implementation of specific predicates for searching XCB's event queue, and a reimplementation of the resource database.

Xlib's interfaces for text internationalization are not provided by XCL. A significant portion of the code size of Xlib is dedicated to translating text strings between various character sets and encodings: XCL deals strictly with glyph rendering using the encoding-neutral interfaces of Xlib. Currently, few Xlib applications and toolkits actually use the internationalized interfaces. Given the decreasing relevance of Xlib's font and text drawing support and the predominance of ASCII encoded text, applications are not expected to need international text rendering functionality from XCL in the future.

Xlib color management is not supported by XCL. The Xlib implementation of this function is by way of integrating an existing color management library: it would be difficult to either duplicate or import this functionality in XCL. In addition, as discussed earlier, few existing applications use the Xlib color management. Many applications use the non-uniform RGB color space provided by Xlib by default. Most applications that do require more sophisticated color management obtain this functionality through toolkit APIs or third-party libraries.

XCL imports some of the caching implementation of Xlib, for example the GCValues cache. In addition, some additional caching is planned, for example an Atom cache. However, substantially less client-side caching is performed by XCL than by the reference Xlib implementation. This decision is a design tradeoff that may need to be revisited as more performance data becomes available.

3.2 Source vs. Binary Compatibility

Many applications and libraries that use Xlib have source code available. In initially specifying requirements for XCL, it was deemed necessary that XCL be at least source-compatible with Xlib, but not necessarily binary-compatible. This weaker requirement leaves open the possibility of sacrificing compatibility of implementation details such as CPP macros and underlying

data structures in order to simplify and ease the implementation.

As it turns out, making XCL largely binary-compatible with Xlib is straightforward. The majority of X applications can use the shared library version of the current XCL implementation as a transparent replacement for the Xlib shared library. Some applications or libraries use obscure features or unpublished interfaces, and may thus need to be recompiled or adapted at the source level.

3.3 Using XCL

XCL is expected to be useful to a variety of different audiences. While each of these target uses is limited, together they cover a broad spectrum of X software development activities. This section enumerates and evaluates some of the expected uses of XCL.

3.3.1 Porting From Xlib to XCB

Applications or toolkits written for Xlib may be ported to XCB in stages using XCL. By simply including `xcl.h` instead of `Xlib.h`, code gains access to the XCB connection structure, and can mix calls to XCB with calls to XCL. Thus software can evolve over time from being entirely Xlib based to being entirely XCB based, claiming the full benefits of XCB.

There are a few cases where the effect of mixing calls to XCL and XCB can be problematic. For example, attempts to manipulate the same graphics context from both XCB and XCL may have unexpected results and lead to client software errors. However, XCL is largely stateless, with most of its state encapsulated by XCB. Most calls to either XCL or XCB should thus produce consistent and predictable effects.

3.3.2 Writing Threaded Code

Xlib is "thread-safe" in the sense that its internal state invariants are protected against simultaneous access by different threads. However, the implementation of thread safety in Xlib is problematic. Internally, a single global lock protects most Xlib state, leading to the possibility of unnecessary interference between unrelated threads and a resulting loss of client efficiency. In addition, the Xlib interface to threading is awkward.

Implementing threaded programs to the XCL interface has a couple of advantages. As noted in the previous section, calls to XCB's more thread-friendly interfaces can be used to replace sections of Xlib code of problematic correctness or efficiency. In addition, the finer-grained locking offers the potential for performance enhancement.

3.3.3 Leveraging X Knowledge

There are a wide variety of books available describing programming at the Xlib level, as well as web sites and other documentation. Since the XCL interface is a large proper subset of the published interface of Xlib, all of these resources can still be used by those wishing to learn to write X applications. Those already familiar with Xlib programming can also avoid a steep learning curve while gaining some of the advantages offered by XCL and XCB.

3.3.4 Understanding X Internals

The implementation of XCL may be instructive to those interested in learning about the inner workings of the X Window System. The division of XCL into an upper layer consisting of code borrowed from Xlib and a lower layer replacing some of the more complex and confusing portions of Xlib with calls to XCB may help to illuminate the structure of Xlib and of X protocol bindings in general.

As another example, the transparent translations to the protocol performed by XCB may provide opportunities for better understanding the runtime behavior of Xlib. For example, consider the Inter-Client Communications Conventions Manual (ICCCM) [Ros] that governs interactions between applications and the desktop environment. The ability of XCB to accurately specify and precisely report the ICCCM interactions of an Xlib window manager running atop XCL may be of great benefit in both validating the window manager and increasing developer understanding of ICCCM.

3.3.5 Developing Toolkits

XCB's latency hiding mechanisms and simple design are expected to be particularly useful to toolkit authors, who can implement optimizations unavailable to Xlib. That expectation applies also to this work. XCL allows toolkit authors to implement XCB-based optimizations in portions of their toolkits needing high performance, while leaving the rest of their Xlib-oriented code base intact. Better yet, toolkits may benefit from further performance gains as XCL and XCB are developed further to support caching and other optimizations.

3.3.6 Accommodating Small Environments

Until XCB becomes widespread on standard systems, applications and toolkits targeted exclusively to XCB will be rare. On the other hand, applications and toolkits targeted to Xlib comprise the bulk of available software. XCL supports these applications by implementing the important portions of Xlib in an extremely lightweight library. The combination of XCL and XCB should provide a platform for legacy X application execution significantly smaller than Xlib. While development should eventually proceed towards new interfaces, support for legacy applications in constrained environments can be a useful feature.

3.4 XCL Influences on XCB

While XCB is largely completed, its development continues in parallel with implementation of XCL. XCB as it currently stands provides a good basis for the XCL implementation. However, in the process of defining XCL, some additional desirable XCB functionality has become apparent.

During the X protocol connection setup phase, a wide variety of per-session server data is sent from the X server to the connecting client. Xlib provides access to this data by way of a plethora of convenience functions. While a clean mechanism for accessing this information has yet to be completed in XCB, a design using accessor functions, including iterators, is currently being implemented. In the meantime, XCB provides direct access to structure accessors for setup data. XCL hides the details of XCB's accessors by copying the data into the XCL `Display` structure in an Xlib-compatible form at connection setup time.

Another aspect of XCB that needs work is its X protocol error handling. In the current implementation, XCB treats X protocol error responses and events similarly, placing error responses in the event queue as though they were events. In addition, XCB does not provide any callback mechanism for error handling, so the only ways to discover that an error has occurred are to process all of the events that precede it in the event queue or to search the event queue using the XCB API. The most significant impact of these XCB design decisions on the design of XCL is that errors often are reported only after dozens of further requests have been processed.

4 XCL Design

XCL copes with two principal APIs. On the client side, it provides the published Xlib API. On the downstream side, it makes calls to XCB. XCL never communicates directly with the X server: all of its requests and all server responses are routed through XCB.

As discussed previously, XCL's overall architecture is a two-layer affair similar to that of Xlib. Reuse of Xlib code in the XCL upper layer eases verification of the requirement that XCL behave identically to Xlib. In fact, one of the implementation strategies used has been to copy the source for the relevant sections of the reference Xlib implementation, remove sections deemed irrelevant to XCL, and replace them with XCB-based equivalent lower layer code. (Another strategy is discussed in section 4.3.)

It may seem that this implementation approach would make the implementation of XCL relatively trivial, but that is not the case. Different sections of Xlib are coupled together tightly: for example, a significant number of utility and convenience functions contain (sometimes slightly modified) copies of code needed to deliver requests to and accept replies from the X server. XCL makes explicit the idea that a lower layer (XCB) handles communication with the X server. XCL also allows other distinct libraries to handle other jobs, such as caching or color management. In doing so, the interface between the convenience functions and the server has been narrowed and standardized, facilitating future code reuse and maintenance.

XCL's current build process uses the Xlib header files installed on the build machine for their definitions of function prototypes and data structures as an aid to matching the behavior specified for Xlib. This means that as long as the same compiler is used, applications built against these header files are sure to be binary-compatible with XCL, as long as they use only supported interfaces. On the other hand, it means XCL is tied to particular versions of particular implementations of Xlib, a difficulty we encountered early in development. A more detailed analysis of Xlib headers is needed to identify those that should be included in the XCL source.

Another downside to reusing Xlib source in XCL is that Xlib source is not always particularly easy to understand. Some defects in XCL have been caused by misinterpreting the semantics of the Xlib implementation. For example, `XPending` and `XNextEvent` were implemented incorrectly during early development of XCL. The reference implementation of Xlib includes functions `_XReadEvents` and `_XEventsQueued`: both may read events from the X server, but `_XReadEvents` always reads one or more events by blocking until some are available, while `_XEventsQueued` reads zero or more without blocking. As this distinction was missed in a superficial reading, it was assumed that `_XReadEvents` could be used for both jobs. As a re-

sult, it was discovered during testing that test applications would freeze once a window was mapped and the event loop was entered.

Since XCL reuses much of the source of an Xlib implementation, there are a large number of small source files in XCL. Xlib has only one or two functions in most of its source files, so that applications statically linked against Xlib can draw in as little code as possible. While this is a sensible plan for the large and monolithic Xlib, it makes less sense for the small and modular XCL implementation. Future work for XCL includes organizing the source appropriately for its environment.

It is worth noting that the license of the reference Xlib implementation (the MIT license) allows development through code reuse. This licensing policy has made the current XCL implementation possible. Had Xlib been developed under a closed-source or limited-availability license, the task of creating XCL would have been much more daunting. Hopefully, making the source to XCB and XCL freely reusable will allow similar opportunities for future developers.

4.1 `XCBConnection` and `Display`

Xlib uses a `Display` data structure to track a single connection to an X server and its state. XCB uses an `XCBConnection` structure for the same purpose. However, these structures have few similarities.

When the XCL `XOpenDisplay` function is called, it in turn calls `XCBConnect` to get an `XCBConnection`. It then sets up a new `Display` structure. In order to support applications that peek into the `Display` structure, XCL copies a number of values from the `XCBConnection` into the `Display`, including the data from connection setup and the file descriptor associated with the server connection. In many applications, this data is part of the required binary interface: although Xlib provides dedicated accessors for this data, the accessors are implemented as CPP macros. It is therefore impossible for XCL to replace the accessors with functions that examine XCB's state directly without breaking binary compatibility. While binary compatibility is not a mandatory requirement of XCL, it is reasonable to provide it when possible: the current architecture permits this.

Early in the development of XCL, an important question arose: how can an `XCBConnection` be associated with a `Display` so that XCL can retrieve the former from the latter? The solution chosen is a standard one: XCL internally manages a structure containing both the `Display` data and a pointer to an `XCBConnection`. The `XCBConnection` pointer is used by internal operations, while the return value of XCL's `XOpenDisplay` is a pointer to the `Display` data.

4.2 XID Types

The X protocol identifies resources such as windows and fonts using XIDs, small integers that are generated by the client rather than the server. In Xlib-based applications Xlib handles generation of new XIDs as needed. XIDs are declared to be 32-bit integers by Xlib, using C `typedef` statements. This has the significant disadvantage of allowing a variety of type-safety errors: because C types are structurally equivalent, it is easy to use a font XID, for example, where a window XID is required. To remedy this problem, XCB uses a distinct structure type for each kind of resource, allowing the C type system to distinguish between them and ensure that XIDs are used in appropriate contexts.

This causes some difficulty for XCL, however: XCL has to deal with and be able to convert between the two systems of XID types. Conversion from XCB-style to Xlib-style types is easy: the single member of an XCB XID structure is of the same 32-bit integer type as the Xlib XIDs are derived from, and so a structure member access (effectively a type coercion) is all that is needed to perform the conversion. Conversion in the other direction, however, is not so simple.

C does not allow anonymous construction of structure-typed values. To work around this difficulty, XCL provides a number of inline functions that declare a structure of an XCB XID type, initialize it with a value of an Xlib XID type, and return the structure by value. In principal, this could be a source of inefficiency. However, as the bit patterns of the types are identical in both systems, a good optimizing compiler ought to be able to inline and then entirely eliminate the conversion code. In fact, the GNU C compiler does a very good job at this.

4.3 Core Protocol Requests

With XCB handling actual communications tasks, XCL simply needs to manage the data it delivers between XCB and the client application. Figure 3 shows the implementation of `XInternAtom`. The implementation delivers an `InternAtom` request to the X server and extracts the atom XID from the reply data.

Most functions in XCL at this point are at least this simple, if not simpler: when caching, color management, internationalization, and other miscellaneous features of Xlib are removed, the core protocol is the largest piece

```
Atom XInternAtom(Display *dpy,
    const char *name,
    const Bool onlyIfExists)
{
    register XCBConnection *c =
    XCBConnectionOfDisplay(dpy);

    Atom atom;
    XCBInternAtomRep *r;

    if (!name)
        name = "";

    r = XCBInternAtomReply(c,
        XCBInternAtom(c,
            onlyIfExists,
            strlen(name), name),
        0);
    if (!r)
        return None;
    atom = r->atom.xid;
    free(r);
    return (atom);
}
```

Figure 3: XCL implementation of XInternAtom.

that remains. Almost all of the code needed to implement the core protocol is supplied by XCB. Functionality only loosely related to the X protocol should be placed in modules separate from XCL itself, to facilitate maintenance and avoid dragging large amounts of dead code along with every X application.

Some XCL functions are even simpler to implement than `XInternAtom`, as illustrated by `XCreatePixmap` in figure 4. As development of XCL has proceeded, common structures became apparent amongst several of the functions that had previously been hand-coded for XCL. In the current implementation, 43 of the roughly 120 requests in the core X protocol are described using a domain-specific language. This language, related to the language used by XCB, describes only the interface to Xlib. A translator to C, implemented in M4 [KR77], reads XCB's interface from XCB's description of the core protocol. Given descriptions of both of the interfaces with which Xlib communicates, the translator can generate the correct code to map parameters between XID type systems and function parameter orderings.

The same benefits expected from XCB's protocol description language (section 2.1) are also expected for XCL's interface description language: maintenance, implementation of new features and extensions, and demonstration of correctness should all be simpler than

```
XCLREQ(CreatePixmap,
    XCLALLOC(Pixmap, pid),
    XCLPARAMS(Drawable drawable,
        unsigned int width,
        unsigned int height,
        unsigned int depth))
```

Figure 4: XCL description of XCreatePixmap.

for hand-coded implementations. The most immediate benefit is that functions generated from this language are known to be implemented to XCB's interface, and can be easily checked against the published Xlib interface specification rather than against the Xlib reference implementation. Future work for XCL thus includes extending this code generation system to implement more requests, including extension requests.

4.4 Protocol Extensions

Since XCB can generate the code for all aspects of the X protocol defined by protocol extensions, implementation of each extension in XCL requires only a proper XCB description of the extension, and any code needed to preprocess requests to or replies from the server.

An example of an extension requiring more than a simple protocol description is the shared memory extension. This extension allows raw data to be transferred through shared memory segments when both the client and the server have access to the same physical memory. XCB can only deal with the X network protocol: shared memory is outside its scope. Thus, an XCL implementation of the shared memory extension would include code to access the shared memory segment, while leaving to XCB the job of exchanging segment identifiers with the server.

4.5 Caching and Latency Hiding

XCL should have as low a latency as reasonably possible from a client call to the desired effect or response. There are two ways to accomplish this goal: hiding the effect of latency from the client application, and removing sources of latency by eliminating high-latency operations.

Caching is an instance of the latter strategy. Little caching is implemented within XCL itself, although graphics context values, for example, are cached using code borrowed from Xlib. Various cache modules will eventually be built directly atop XCB: as these become available XCL will be modified to use them. Until then, most requests through XCL will produce protocol requests to the X server. This can substantially increase

latency in client applications under XCL: fortunately the impact on client applications in current common environments is expected to be minimal.

XCB has latency hiding functionality, and it is desirable to extend this effect to XCL. Unfortunately, Xlib's interface is generally not conducive to this effort. In most cases where a reply is expected, Xlib's interface requires the caller to block until the reply arrives from the X server. However, there are some cases where the interface allows for a potentially large number of requests to be processed in parallel.

`XInternAtoms` (the plural here is important) is one such case: given a list of atom names to map to atom IDs, it sends requests for all of the atoms to the server, and then waits for each of the replies. The Xlib version of this code, using the async-handler interface discussed in section 2.3, is significantly more complex than the XCL version built on XCB. XCL's `XInternAtoms` implementation contains considerably less code than Xlib's. (Although, to be fair, this is also because the atom cache has been decoupled from the library and is not yet implemented).

4.5.1 Request Marshaling

Xlib's ability to marshal several independent client drawing requests (such as line drawing requests) into a single protocol request is of minor importance for performance on core protocol requests, at least on modern hardware. However, more recent X extensions, particular the Render extension [Pac01a], depend on request marshaling for performance. In these cases, marshaling opportunities are frequent, and failure to do so may be expensive.

XCB provides request marshaling using a mechanism transparent to clients. In XCB, the ability of a type of request to be marshaled is considered an attribute of the request, and is specified in the same protocol description language as is used to define the binary protocol encoding. Care has been taken to ensure that extensions and the core protocol may be described with the same language, so request marshaling is available to any extension with no more effort on the part of the extension implementor than an extra line of markup.

Because of the transparency of XCB marshaling, the primitive drawing requests in XCL have been trivial to implement, while still performing competitively with their Xlib counterparts. In fact, because only Xlib's `XDrawLine` (singular) and similar functions marshal in the reference implementation, while `XDrawLines` (plural) and others do not, some applications could theoretically experience performance gains just by re-linking with XCL: this could be especially important over low-bandwidth or high-latency links, where marshaling in the core protocol has the most effect. Since marshaling is an intrinsic capability of XCB and not of XCL, all applications built on XCB gain the performance benefits, not just clients built on XCL.

4.6 Threaded Clients

XCL's support for threaded applications is about as complete as the Xlib API will allow. Unfortunately, certain race conditions are still possible, as discussed in section 2.4. Clearly, XCL cannot ensure the correctness of threaded programs written to the Xlib interface. However, it can improve performance for multi-threaded applications.

Xlib uses a single lock to protect the entire contents of its `Display` data structure, meaning that for most applications all calls into Xlib are serialized. However, there is no reason in principle to disallow multiple threads to access disjoint portions of the internal state simultaneously: while one thread is accessing the event queue, another could be sending a request to the server, and a third could be querying the GC cache.

Currently, XCL allocates a single lock for the entire `Display` as Xlib does, but for protocol requests that lock is unused. XCB provides a separate thread-safety mechanism for its `XCBConnection` data that ensures that protocol requests and responses are handled correctly. Any caches implemented atop XCB in the future can (and should) handle their own data structures in a thread-safe fashion.

By leveraging XCB's thread-safety mechanisms, XCL provides fine-grained locking. This may allow multi-threaded applications to take better advantage of the resources of a system than the Xlib implementation would permit.

4.7 Events and Errors

Event and error handling represents a fairly significant portion of the implementation of XCL. Not coincidentally, this portion of the implementation is also one of the least well-specified and understood pieces of Xlib. Several aspects of event and error handling warrant further consideration.

4.7.1 The XCL Event Queue

XCB's event queue is accessed through simple traversal functions. However, this narrow interface is sufficient

for the XCL implementation of Xlib's event search interfaces. Xlib's predicated event retrieval functions may all be implemented as traversals of the event queue, evaluating each event against some arbitrary predicate.

Internally, XCB has a generic linked list implementation that it uses to track several distinct types of information, including a list of replies expected from the server. All users of these internal XCB lists must be able to search for particular items: thus, when the predicated event queue code was added, its implementation added almost no new code to XCB. All of XCB's event queue management implementation combined amounts to only 20 lines of code.

Throughout the design and implementation of XCB and XCL, significant decreases in code size and improvements in maintainability have resulted from the use of modular, layered architecture. Separating list implementations from event queue maintenance and event queue maintenance from XCL predicate implementations provides a nice example of this phenomenon. The modular, layered design provides flexibility, permitting such improvements as reimplementation of internal list interfaces atop new data structures, or pooling of list nodes for reuse. This can be accomplished without modifications to the rest of XCB or XCL, providing improvements transparently to all XCB and XCL client applications.

4.7.2 X Protocol Errors

When XCB reports to XCL that an X protocol error has been received, XCL passes the error off to an error handler as required by the Xlib API. The default error handler displays a (somewhat) human-readable description of what went wrong and terminates the application: however, the handler may be replaced by the client program with an arbitrary client function. This mechanism is awkward to use: as a result, most existing Xlib applications exhibit poor behavior in response to protocol errors. Unfortunately, it is not clear how to address that problem without changing the Xlib error handling API.

4.7.3 XCL and XCB Internal Errors

The reference implementation of Xlib performs various error checks on data coming both from Xlib's callers and from the X server. XCB simply delivers whatever data it is given. The primary benefit of this approach is that XCB may deliver the data faster. An interesting side benefit is that XCB allows for the creation of certain kinds of tools for testing X servers by delivering bad input. Naturally, the trade-off is that XCB does not particularly help a developer debug a faulty client or server.

In most cases, XCL and XCB have identical failure modes to Xlib, and XCL can return status codes identical to Xlib's when an XCL or XCB error occurs. However, in a small number of instances, XCL can fail in ways that Xlib could not. First, XCL will detect failure in situations that Xlib does not. Second, there may be opportunities for the XCL implementation to fail that were not present in the Xlib implementation (as in the `XDrawString` example below). Unfortunately, Xlib has no particularly well-structured or application-honored mechanism for reporting errors. The current XCL implementation makes reasonable efforts to sensibly handle and report internal errors within the bounds of the Xlib API semantics.

For example, `XDrawString` has to break the string given to it into 254 character chunks, with a two-byte header per chunk. Xlib uses the `Display`'s output buffer directly, while XCL uses `malloc` to create a temporary buffer, subjecting XCL to potential out-of-memory errors. In XCL's current implementation, we call the Xlib-compatible `_XIOError` handler when `malloc` fails.

Despite the fact that even the Xlib implementation of `XDrawString` can fail (if the string provided has a non-positive length), the Xlib implementation always returns success. It would be nice to develop better error handling and reporting mechanisms for these cases within the constraints of the Xlib API.

5 Results

The implementation of XCL is not yet quite complete. However, we have some promising preliminary results.

5.1 RXVT on XCL

The XCL development model includes an initial iteration implementing enough of Xlib to support a single reasonable-sized Xlib application. This is intended to serve as proof of concept, and as validation and evaluation of the design and implementation. The application we chose was the `rxvt` [rxv] terminal emulator, a more modern replacement for `xterm`.

The `rxvt` terminal emulator is related to the standard `xterm` utility in the same way that XCL is related to Xlib. According to the manual, `rxvt` is "intended as an xterm replacement for users who do not require features

such as Tektronix 4014 emulation and toolkit-style configurability", with the benefit that "`rxvt` uses much less swap space".

This similarity of purpose made `rxvt` an attractive initial target for XCL. In addition, `rxvt` is a reasonably powerful X application that performs very useful work. Finally, `rxvt` exercises a large portion of the core X protocol, including text, rendering and events.

XCL is currently complete enough that `rxvt`, without any source code changes, may be linked against XCL rather than Xlib and will run correctly. In fact, the current stable and development versions of `rxvt` can be compiled against Xlib and then correctly executed against the XCL library binary. The first iteration of XCL development is therefore complete and successful. The remaining work is expected to be largely straightforward, if time-consuming.

5.2 Gnome `gw`: GDK on XCL

As mentioned in section 3.3.5, one expected use of XCL is with toolkits originally written for Xlib. To test the feasibility of this use, we selected one of the simpler Gnome/GTK+ applications for trials on early versions of XCL. `gw` is a graphical version of the venerable Berkeley "`w`" command used to list the users currently logged into a Unix system.

The `gw` user interface contains a button, a list widget with a scrollbar, and a dockable menu bar. We found that these widgets need only a small subset of Xlib's functionality. Getting the basic interface to run on XCL was a simple matter: scrollbars scroll, buttons click, menus drop down, and menu bars undock and re-dock. GTK+ and Gnome/GTK+ applications are implemented using a window system independent layer known as GDK. Since the XCL functionality necessary to get `gw` working covers a large portion of the Xlib API subset used by GDK, there is reason to believe that a substantial fraction of the work needed to port GTK+ in its entirety is already completed.

XCL is not perfect yet: some aspects of current `gw` operation atop XCL are visibly wrong, or trigger X protocol errors that shut down the application. `XPutImage` and `XSendEvent`, among others, have been sources of trouble. While fixing these problems may require substantial debugging effort, no major technical barriers are expected.

5.3 Size and Performance

While the implementations of XCL and XCB are still subject to change, it is nonetheless useful to take at least a first look at the size and performance of the current system. Here are some preliminary metrics:

5.3.1 Size

The size metric used is kilobytes of code in the text section of the compiled object file. The text section needs to be stored both on disk and in memory, both of which are scarce on small systems. In addition, if the entire text section can fit into the L2 cache of the target system (typically between 64kB and 512kB on modern machines), a significant performance improvement might be possible for typical applications making a mix of calls into the library. Including data and BSS size would not significantly impact the reported measurements. All numbers are produced by examining the text sections of either object (`.o`) files or statically-linked library (`.a`) files on an Intel x86 architecture machine. Our tests show that use of the optimizer of the compiler is a major factor in the size of both XCB and XCL. The numbers that follow for XCB and XCL are produced by analyzing the output of "`gcc -O2`": the unoptimized output is nearly 50% larger.

82% of the 79 object files currently comprising XCL have text sections smaller than 512 bytes. The total text section size of XCL is 28kB. XCB adds another 27kB, for a total of 55kB.

Xlib, compiled from 417 source files, has a text section size of 658kB. (According to Jim Gettys [Get01], much of this is data used as lookup tables for internationalization.) The subset of Xlib providing roughly the same functionality as XCL has about 115kB of text. This is more than twice the size of XCL combined with all of XCB. In addition, it would be quite difficult to build a shared library version of Xlib containing just this functionality.

5.3.2 Performance

The XCL version of `rxvt` currently seems to have text display performance comparable to that of the Xlib version. Executing "`cat /usr/share/dict/words`" on a 1GHz Mobile Pentium III running Linux yields runtimes of about 1.3 seconds for `rxvt` 2.7.8 when linked with either Xlib or XCL. Results are similar on a 700MHz Athlon, although XCL consistently exhibits about a 5% speed advantage on this platform for this benchmark.

It is notable that, at about 30,000 scrolled lines per second, `rxvt` is more than acceptably fast for any reasonable use. Indeed, this is the normal case for X appli-

cations on modern hardware: it is extremely unusual to find an application whose performance is limited by interaction with an X server on the same host. In addition, Xlib is believed to provide near-optimal performance in most situations. For these reasons, performance enhancement is not a primary goal of XCL.

6 Related Work

We know of no other efforts to design X protocol client libraries in C. There have been some independent efforts to write such libraries for a variety of other languages including Java [O'N02, Tse01], Common Lisp [SOC+89], Smalltalk [Byr], and ML [GR93], as well as more exotic languages such as Python, Erlang, and Ruby. These efforts have concentrated largely on providing natural bindings for their target language, with performance, size, and compatibility with the Xlib API being at most secondary targets.

The Nano-X [Hae01] GUI environment has some interesting parallels to XCL. Nano-X is targeted at lightweight and embedded systems, and provides an API roughly comparable to that of Xlib. However, Nano-X supports a variety of underlying rendering systems, only one of which is the X core protocol. The X protocol support of Nano-X is intended primarily for development and debugging, and is not intended as a principal API for normal application use.

Over the years, Jim Gettys and others have put a significant amount of effort into improvements to the Xlib implementation. Much of this work has been to increase Xlib functionality. More recently, Gettys has put some effort into reducing the size of Xlib for use with Linux on the Compaq iPaq hand-held computer.

The authors' work on XCL was inspired to some extent by an award-winning program from the 1991 International Obfuscated C Code Contest [NCSB02]. This remarkable 1676 character C program by David Applegate and Guy Jacobson runs Conway's "Game of Life" cellular automaton on an X root window. Its small size and remarkable performance provided a powerful hint of what is possible with clever coding.

7 Status and Future Work

The XCL implementation is well underway: as noted in the previous section, a useful test application (`rxvt`) links with and runs on XCL. When all Xlib protocol wrapper functions have been implemented in XCL, many more applications are expected to run without modification.

XCL is dependent on the XCB library implementation, which is nearly complete. In its current form, it provides the majority of the functionality needed for the XCL implementation. Some work remains. For example, accessors need to be constructed for variable-element-length lists such as those sent from the X server on connection setup, and XCB error handling needs further exploration.

Allowing a wider variety of applications to run on XCL is the immediate focus as the project continues. One way to quickly support a large number of applications is to support one or more GUI toolkits: preliminary results for GTK+ and Gnome are promising, and we plan to further research this possibility in the near future. As discussed in section 3.3.5, migrating toolkits to XCL may be a good first step in eventually migrating them to XCB.

8 Conclusions

XCL, in conjunction with XCB, represents the first real alternative to Xlib since Xlib's inception more than 15 years ago. As both are open source, the X development community can examine both for their merits and produce software that is useful for a wide variety of platforms and applications.

Much of the hype surrounding the development of freely-available UNIX software in recent years springs from the idea that open development and the use of freely-available source materials can produce high-quality software products that can then be used to bootstrap future development in this style. XCL provides a nice example of this phenomenon, as well as being a useful tool in its own right. In the immortal words of Hannibal Smith, "I love it when a plan comes together."

Availability

Current implementations of XCL and XCB are freely available under an MIT-style license at `http://xcb.cs.pdx.edu/`.

Acknowledgements

The authors gratefully acknowledge the advice and assistance of Keith Packard, Jim Gettys, and other X contributors in the design and analysis leading up to XCL. Andy Howe has played a major part in the implementation and testing of XCL. Finally, Chris Demetriou was invaluable throughout the tough task of shepherding this paper.

References

[AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Digital Press, Bedford, MA, 1990.

[Ber95] David T. Berry. Integrating a color management system with a Unix and X11 environment. *The X Resource*, 13(1):179–180, January 1995.

[Byr] Steve Byrne. *GNU Smalltalk Version 1.1.1 User's Guide*. Web document. URL `http://www.cs.utah.edu/dept/old/texinfo/mst/mst_toc.html` accessed April 3, 2002 09:12 UTC.

[Dal01] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, 2001.

[Get01] Jim Gettys, 2001. Personal communication.

[Gil99] Graeme Gill. Icc file I/O, 1999. Web Document. URL `http://web.access.net.au/argyll/color.html` accessed April 11, 2002 09:25 UTC.

[GR93] Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1, pages 61–80. John Wiley & Sons, 1993.

[Hae01] Greg Haerr. *Nano-X Reference Manual*, January 2001. Web document. URL `ftp://microwindows.org/pub/microwindows/nano-X-docs.pdf` accessed April 3, 2002 21:00 UTC.

[KR77] Brian W. Kerninghan and Dennis M. Ritchie. *The M4 Macro Processor*. AT&T Bell Laboratories, 1977. Unix Programmer's Manual Volume 2, 7th Edition.

[lcm] Little CMS. Web Document. URL `http://www.littlecms.com/` accessed April 11, 2002 09:23 UTC.

[MS01] Bart Massey and Jamey Sharp. XCB: An X protocol C binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[NCSB02] Landon Curt Noll, Simon Cooper, Peter Seebach, and Leonid A. Broukhis. International Obfuscated C Code Contest, 2002. Web document. URL `http://www.ioccc.org/` accessed April 8, 2002 05:58 UTC.

[O'N02] Eugene O'Neil. XTC: the X Tool Collection, April 2002. Web document. URL `http://www.cs.umb.edu/~eugene/XTC/` accessed April 3, 2002 07:33 UTC.

[Pac01a] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[Pac01b] Keith Packard. The Xft font library: Architecture and users guide. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[Pen99] Havoc Pennington. *GTK+/Gnome Application Development*. New Riders Publishing, 1999.

[Ros] David Rosenthal. Inter-Client Communication Conventions Manual. In [SGFR92].

[rxv] RXVT. Web document. URL `http://sourceforge.net/projects/rxvt/` accessed April 8, 2002 5:56 UTC.

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SGFR92] Robert W. Scheifler, James Gettys, Jim Flowers, and David Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, and XLFD*. Digital Press, third edition, 1992.

[SOC+89] Robert W. Scheifler, LaMott Oren, Keith Cessna, Kerry Kimbrough, Mike Myjak, and Dan Stenger. *CLX Common Lisp X Interface*, 1989.

[Tse01] Stephen Tse. Escher: Java X11 library, 2001. Web document. URL `http://sourceforge.net/projects/escher` accessed April 3, 2002 20:58 UTC.

[xfr] The XFree86 project. Web document. URL `http://www.xfree86.org` accessed April 8, 2002 05:54 UTC.