FEBRUARY 1991

WRL
Research Report 91/1

Writing Fast X Servers for Dumb Color Frame Buffers

Joel McCormack

Western Research Laboratory 100 Hamilton Avenue Palo Alto, California 94301 USA

Writing Fast X Servers for Dumb Color Frame Buffers

Joel McCormack

February 1991

d i g i t a l **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

Processor speeds are improving faster than memory access speeds. The current generation of RISC processors can perform many 2-dimensional graphics operations at or near memory or bus bandwidth speeds, rendering specialized hardware unnecessary for all but the most demanding applications. This paper describes the DECStation 3100 and DECStation 500/200CX color frame buffer hardware and several of the graphics algorithms used in the X server implementation. Measured performance numbers are presented and compared to memory bandwidth speed, and the most important contributors to the frame buffer's performance are summarized.

This report is a preprint of a paper that will appear in the special X11 issue of *Software—Practice and Experience* 20(2), and supersedes WRL Technical Note TN-9.

1. Introduction

Color workstations have typically had a screen size of 1024 to 1280 columns by 768 to 1024 rows, 8 bits per pixel, and a processor capable of executing one to three million instructions per second. Color workstations are traditionally equipped with special-purpose graphics accelerators in order to achieve reasonable performance.

The DECstation 3100[1] supports a 1024x864x8 color display, but uses no special graphics hardware. The MIPS R2000 processor[2] paints directly into a simple frame buffer. We originally hoped to obtain graphics performance equivalent to the VAXStation 3100, which is rated at about three million instructions per second and has two specialized graphics chips built using old technology. We ultimately found that the frame buffer offers graphics performance roughly two to four times better than the chip set. In fact, for many operations the frame buffer offers performance comparable to modern graphics accelerators.

The DECstation 5000/200CX builds upon experience with the DECstation 3100. Like its predecessor, the DECstation 5000's MIPS R3000 processor also paints directly into the frame buffer. However, circuitry to exploit ‘‘page-mode’’ access to the video RAMs more than doubles available write bandwidth.

This paper discusses the video memory system and some useful characteristics of the MIPS instruction set. Graphics algorithms for painting lines and text and for copying data illustrate how the server exploits a fast processor, and show the memory bandwidths these algorithms are capable of using. The most desirable characteristics of an instruction set and memory system are summarized, and the relative merits of frame buffers and graphics accelerators are discussed.

[1]DECstation 3100, DECstation 5000, VAX, VAXStation 3100, and TURBOchannel are trademarks of Digital Equipment Corporation.

[2]MIPS, R2000, and R3000 are trademarks of MIPS Computer Company. The ‘‘MIPS instruction set’’ refers specifically to the R2000 and R3000 CPU chips, and unless otherwise noted refers to the R6000 chip set and future MIPS CPUs as well.

2. Frame Buffer Hardware

The DECstation 3100 color frame buffer is arranged as 1024 scanlines of 1024 8-bit pixels. The display hardware uses each pixel as an index into a 256-entry colormap, which delivers 24 bits of RGB data to the digital-to-analog video converter. The smallest address in a scanline (last ten bits are all 0) displays at the leftmost edge of the screen; the largest address in a scanline (last ten bits are all 1) displays at the rightmost edge of the screen.

Only the first 864 scanlines are actually displayed. In systems with hardware graphics accelerators, the extra 160 kilobytes would be used as an off-screen cache for font information or pixmaps. We don’t bother using this memory: it offers no performance advantage over main memory, and would add to the complexity of memory management.

Main memory and video memory share the same internal bus and timing logic. The processor can read or write any contiguous group of one to four pixels in a word with one memory access. Measured memory access times are 400 nanoseconds for a write, 410 nanoseconds for a read, and 760 nanoseconds for a read followed by a write. Sustained write bandwidth is 10.0 megabytes per second; sustained copy bandwidth is 5.3 megabytes per second.

The frame buffer has an 8-bit planemask, which is replicated 4 times across the word. A 1 in the planemask allows the corresponding destination bit to be overwritten, a 0 in the planemask leaves the destination bit unchanged. This allows the server to avoid a read/modify/write cycle when a graphics request specifies a planemask with one or more 0 bits.

The DECstation 5000/200CX frame buffer memory is arranged identically to the DECstation 3100. Video memory is not tightly coupled to the processor, but is accessed across the TURBOchannel I/O bus. Most writes are fast, as the frame buffer includes ‘‘page-mode’’ logic. The first write to a VRAM page uses a normal memory cycle, measured at 360 nanoseconds. Subsequent writes to the same page use a fast page-mode cycle, measured at 180 nanoseconds. (One VRAM page spans two scanlines of the frame buffer.) Sustained write bandwidth is 22 megabytes per second. A lack of board real-estate prohibited the frame buffer from performing page-mode writes in the 120 nanosecond minimum transaction time supported by the bus.

Measured read times are 610 nanoseconds for the first access to a VRAM page, and 440 nanoseconds for page-mode reads. The slow read times result in part from the the TURBOchannel bus design, which has a minimum read cycle of 160 nanoseconds, and in part from the R3000 chip design, which optimizes multiple-word reads from the cache over single-word reads. Sustained copy bandwidth is 6.0 megabytes per second, just a little better than the DECstation 3100.

The page-mode logic uses too much board space to also include planemask logic. Software techniques described below lessen the need for a hardware planemask, and so trading the planemask for page-mode memory access improves the performance of most applications. There are a small class of painting operations, however, that execute more slowly on the DECstation 5000/200CX than on the older DECstation 3100.

3. CPU Hardware

The DECstation 3100 uses a MIPS Computer System R2000 processor [6] with a 60 nanosecond clock. This processor is generally rated at 12-14 times the speed of a VAX 11/780 on integer operations. The DECstation 5000 uses an R3000 processor with a 40 nanosecond clock; this processor is about 20 times the speed of a VAX 11/780.

On both machine, the processor has a 64 kilobyte direct-mapped instruction cache, and a 64 kilobyte write-through direct-mapped data cache. We assume that small amounts of code and small data tables are cached if they have been accessed recently. Information from the data cache is not available to the instruction immediately following a load, but the MIPS instruction scheduler usually fills this load-delay slot with useful work.

On the DECstation 3100, writes to both cached and uncached memory go through a 4-word deep write buffer. The buffer is useful for unrolling loops: the processor executes several store instructions, then performs loop overhead ‘‘for free’’ while the write buffer drains to memory. The write buffer also coalesces sequential writes to contiguous bytes, but the algorithms described below usually write contiguous bytes in one memory access anyway.

The DECstation 5000 has a 6-word deep write buffer that does not coalesce bytes. Since coalescing rarely helps the X server, and benchmarks showed that it wouldn’t help normal applications much either, we got something useful instead: two extra words of write buffer.

The MIPS chips can be configured for ‘‘big-endian’’ or ‘‘little-endian’’ byte order. For compatibility with the VAX architecture, all DECstations are little-endian: byte 0 of a word is the least significant 8 bits, byte 3 of a word is the most significant 8 bits. All algorithms described are little-endian, but would work nearly as well on a big-endian CPU with minor modifications.

4. Addressing the Frame Buffer

In theory, the MIPS memory addressing architecture offers four possibilities for accessing the frame buffer: addresses can be physical or virtual, and the referenced data can be cached or uncached.

In reality, physical addressing is available only in the protected kernel mode; the processor traps a user program attempting to use a physical address. The X server can use physical addresses if the server is incorporated into the operating system kernel, but this makes it hard to debug and update the server, and reduces the reliability of the kernel. (A previous attempt to put the X10 server into the kernel by another group made these facts painfully obvious.) As a result, the server runs in user mode, and uses virtual addresses to access the frame buffer; this causes some performance degradation that is discussed later.

We don’t cache accesses to frame buffer memory for several reasons. First, the displayed portion of the screen is 864 kilobytes, but the data cache is only 64 kilobytes. Painting the contents of even a small window would saturate the cache with screen data, pushing aside large amounts of useful data already in the cache. Second, nearly all accesses to the color frame buffer are write instructions. Finally, we measured performance of cached and uncached access to the monochrome frame buffer, which has 108 kilobytes of screen memory mapped onto 32 kilobytes of the data cache. Unlike the color code, monochrome painting code often performs several read/modify/write cycles on the same word. Even with these advantages, benchmarks showed that caching improved monochrome painting at most 10%, and often degraded performance. The results were so disappointing we didn't bother to measure cached performance for the color server.

5. Useful MIPS Instructions

The MIPS architecture has several instructions, particularly loads and stores for data smaller than a 32-bit word, that are not found in all RISC architectures. This section describes the semantics of these instructions; later sections show how the server uses them to avoid various read/modify/write idioms, and thus increase performance.

This section describes only the store instructions; there are corresponding load instructions in each case. Also note that all load and store instructions contain a 16-bit signed byte offset, which makes unrolled loops quite efficient. The server uses a small constant offset for each unrolled source and destination reference, then at the end of the loop increments both pointers by the total number of bytes processed.

`Store Byte` (`sb`) stores the least significant byte of a register into the specified destination. `Store Halfword` (`sh`) stores the least significant two bytes of a register into the specified destination, which must be halfword-aligned (the lowest bit of the address must be 0). `Store Word` (`sw`) stores all four bytes into the specified destination, which must be word-aligned (the low two bits of the address must be 00).

`Store Word Right` (`swr`) writes the least significant bytes of the source register to the most significant bytes of the destination word. It writes one to four bytes, depending upon the alignment of the destination address. Conceptually, it starts at the lowest byte in the source register, copying byte by byte until it reaches the highest byte of the destination word. If the destination address is word-aligned, `swr` writes all four bytes, and acts just like a `Store Word` instruction. If the destination address points to the highest byte of a word (the low two bits of the address are 11), `swr` writes the lowest byte of the register into the highest byte of the word. Figure 1 shows an example of `Store Word Right` in action.

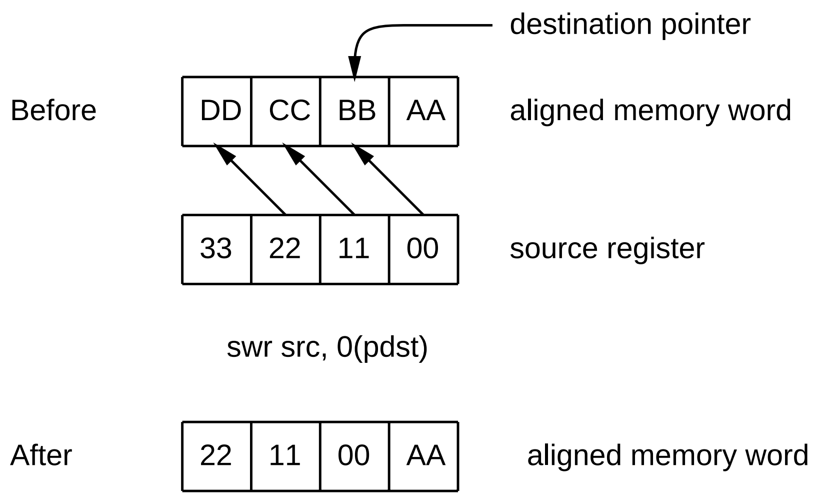


Figure 1: The Store Word Right instruction

`Store Word Left` (`swl`) writes the most significant bytes of the source register to the least significant bytes of the destination word. It writes one to four bytes, conceptually starting at the highest byte in the source register, copying byte by byte until it reaches the lowest byte of the destination word. If the destination address points to the highest byte of a word, `swl` writes all four bytes of the register to the destination word. If the destination address points to the lowest byte of a word, `swr` writes the highest byte of the register to the lowest byte of the word. Figure 2 shows an example of `Store Word Left` in action.

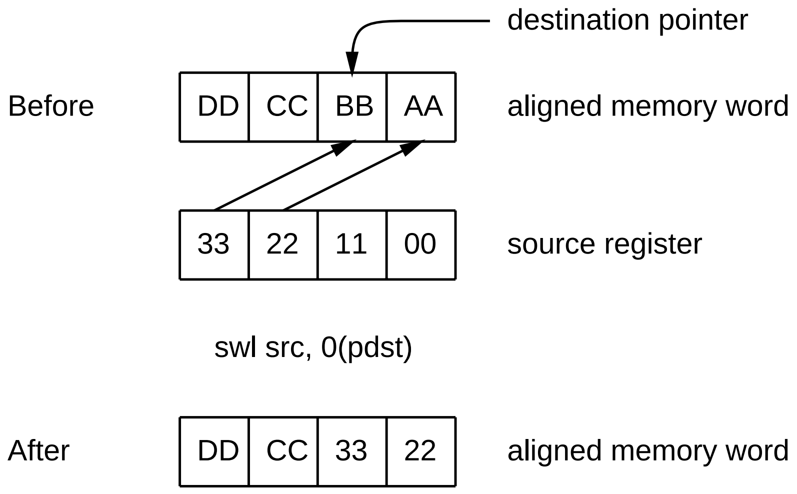


Figure 2: The Store Word Left instruction

While it is easy to use type declarations in the C language to gain access to the byte and halfword instructions, there is no direct way to use the `Load/Store Word Left/Right` instructions from C. We defined four assembly-language procedures that execute two instructions: one `Load/Store Word Left/Right` instruction, and one return instruction. Though procedure overhead is apparently minimal, using a procedure call substantially reduces the effectiveness of the register allocator. The allocator doesn't know how innocuous the assembly procedure actually is, and so makes inefficient use of the caller-saved registers. Still, in most cases this is the fastest alternative available. The few painting routines written in assembly language use `swl` and `swr` directly with no negative impact on the human register allocator.

6. X Graphics Functions and the Planemask

X imaging can be broken into three independent problems: computing the shape of the image to paint, computing the source pixels to paint, and finally combining the source pixels with the existing destination pixels. Conceptually, the shape of an image is broken down into a list of ‘‘spans,’’ where each span is a contiguous sequence of pixels on a scanline. Each span is converted into a (possibly non-contiguous) sequence of source pixels according to the specified fill style, and the source pixels are then combined with the destination pixels according to the specified graphics function (‘‘raster op’’). In reality, many algorithms fill each span immediately, and may also precompute information common to all spans in an object. This section describes how to deal with graphics functions; the two subsequent sections describe how to paint the various fill styles efficiently and how to handle problems specific to common X drawing requests.

The X protocol allows a source pixel and a destination pixel to be combined using any of the 16 possible 2-operand Boolean functions. If a pixel contains more than one bit, the same graphics function applies to all bits (or ‘‘planes’’) of the pixel. Table 1 shows the name and definition of each graphics function.

Function	Definition
Clear	0
And	src AND dst
AndReverse	src AND (NOT dst)
Copy	src
AndInverted	(NOT src) AND dst
NoOp	dst
Xor	src XOR dst
Or	src OR dst
Nor	NOT (src OR dst)
Equiv	NOT (src XOR dst)
Invert	NOT dst
OrReverse	src OR (NOT dst)
CopyInverted	NOT src
OrInverted	(NOT src) OR dst
Nand	NOT (src AND dst)
Set	1

Table 1: X11 graphics functions

The protocol also supports a planemask: wherever the planemask has a 1 bit, the result of the graphics function is written to the corresponding destination bit; wherever the planemask has a 0 bit, the corresponding destination bit is left unmodified. If `FUNC` represents one of the 16 Boolean functions, planemask semantics can be expressed as:

```
    ((src FUNC dst) AND planemask) OR (dst AND (NOT planemask))
```

Note that if every bit in the planemask is 1, this reduces to the much simpler equation:

```
    src FUNC dst
```

While the DECstation 3100 frame buffer has a hardware planemask, the DECstation 5000/200 frame buffer does not. Neither frame buffer has logic to implement the 16 graphics functions. And main memory, where off-screen images (pixmaps) reside, has no special logic for graphics functions or for the planemask. Therefore, we needed an efficient software implementation of the X11 graphics functions and the planemask.

A simple-minded approach might implement each graphics painting routine 32 times: once for each of the 16 graphics functions with complete planemask semantics, and once for each with the simplified all-1 planemask semantics. This obviously uses a lot of code memory.

Another technique is dynamic code generation, in which the inner loops of painting algorithms are generated at run-time. While this has worked well in simple graphics systems, X11 allows too many combinations of shapes, fill styles, and graphics functions. The color server has nearly three hundred calls to the low-level painting macros, many of them in the depths of complex rendering algorithms. Dynamically generating code for all these cases is impracticable.

We instead reduced the number of different graphics functions by deriving more general functions. We kept the following goals in mind:

- Make the most commonly used functions (especially `Copy`) run fast.
- Avoid reading the destination pixels for the functions `Copy`, `Clear`, `Set`, and `CopyInverted`.
- Incorporate planemask semantics into the derived functions when possible, rather than using the separate planemask equation above.
- Exploit the fact that most painting uses a known source value—the foreground or background pixel.

Just to get the DECstation 3100 server running, we initially compiled each painting routine three times: once for `Copy`, once for `Xor`, and once for everything else. `Copy` and `Xor` were implemented as defined, while the remaining 14 functions used a `switch` statement to branch to the appropriate code each time the routine needed to paint. This scheme had several drawbacks: it didn't incorporate planemask semantics, any request that used a function other than the `Copy` or `Xor` painted quite slowly, and the code files generated using the `switch` statement were four to eight times large than files for the `Copy` and `Xor` cases.

Our next release used parameterized functions to avoid the deadly `switch` statements. The 16 functions were reduced to four functions for filling operations (in which a known foreground and background are painted), and six functions for `CopyArea`, `PutImage`, and tiling (in which the source pixels are fetched from some area of memory, so it is hard to preprocess them into a useful form). The fill functions required from 0 to 7 logical operations; the `CopyArea` family of functions required from 0 to 8 logical operations. Though we compiled each source file more times than in our first scheme, the total code size was smaller, and painting with any of the 14 ‘‘other’’ functions was substantially faster.

I told Keith Packard at MIT about our results, and he immediately attacked the problem from the other end. Our derived functions fell into three general forms and showed a distinct lineage from the original X11 functions. Keith knew that reduction to three functions was possible, and so tried to find a single logic equation to express all 16 functions. His first generic equation for fill operators was:

```
    ((dst XOR xorbits) AND andbits) OR orbits
```

where the parameters `xorbits`, `andbits`, and `orbits` were derived from the known source pixel. He soon reduced this to:

```
    (dst AND andbits) XOR xorbits
```

Since one general equation can implement all 16 functions, each plane in the pixel can have a different function applied to it by the appropriate selection of `andbits` and `xorbits`. Forcing the `NoOp` function (`andbits` = 1, `xorbits` = 0) for each plane in which the planemask contains a 0 bit automatically implements planemask semantics with no additional logic operations.

The general function can be split into more specific functions in order to avoid reading the destination when it isn't used, and to avoid logical operations that accomplish nothing. We use the three derived fill functions `DFcopy`, `DFxor`, and `DFgeneral`. Table 2 shows how `andbits` and `xorbits` are derived from the known source pixel (the specified foreground or background), and which specific derived function is actually used. To add planemask semantics, we force a `NoOp` in planes with a 0 by computing

```
    andbits = andbits OR (NOT planemask)
    xorbits = xorbits AND planemask
```

X11 Function	Fill Function	andbits	xorbits
Clear	DFcopy	0	0
And	DFgeneral	src	0
AndReverse	DFgeneral	src	src
Copy	DFcopy	0	src
AndInverted	DFgeneral	~src	0
NoOp	DFxor	1	0
Xor	DFxor	1	src
Or	DFgeneral	~src	src
Nor	DFgeneral	~src	~src
Equiv	DFxor	1	~src
Invert	DFxor	1	1
OrReverse	DFgeneral	~src	1
CopyInverted	DFcopy	0	~src
OrInverted	DFgeneral	src	~src
Nand	DFgeneral	src	1
Set	DFcopy	0	1

Table 2: Reduced graphics functions for fill painting request

Table 3 shows the definitions of the specific fill functions, which are all of the form

```
    Function(dst, andbits, xorbits)
```

Fill Function	Definition
DFcopy	xorbits
DFxor	dst XOR xorbits
DFgeneral	(dst AND andbits) XOR xorbits

Table 3: Definitions of the three fill functions

Note that `DFcopy` cannot implement planemask semantics if any bit in the planemask is 0. In this case, we fall back to the `DFgeneral` code. (`DFcopy` and `DFxor` are identical to our previous derived functions, but Keith's `DFgeneral` function is far simpler than our previous `DFandpix` and `DFcopypix` functions.)

`CopyArea`, `PutImage`, and tiling are fundamentally different from all other X11 painting operations. In general, the values being copied are not known, and so we cannot preprocess the source the way that the fill functions do. The general `CopyArea` equation is a recursive expansion of the fill equation:

```
    (dst AND ((src AND andbits1) XOR xorbits1))
         XOR ((src AND andbits2) XOR xorbits2)
```

X11 Function	Copy Function	andbits1	xorbits1	andbits2	xorbits2
Clear	DCgeneral	0	0	0	0
And	DCgeneral	1	0	0	0
AndReverse	DCgeneral	1	0	1	0
Copy	DCcopy	0	0	1	0
AndInverted	DCgeneral	1	1	0	0
NoOp	DCxor	0	1	0	0
Xor	DCxor	0	1	1	0
Or	DCgeneral	1	1	1	0
Nor	DCgeneral	1	1	1	1
Equiv	DCxor	0	1	1	1
Invert	DCxor	0	1	0	1
OrReversed	DCgeneral	1	1	0	1
CopyInverted	DCgeneral	0	0	1	1
OrInverted	DCgeneral	1	0	1	1
Nand	DCgeneral	1	0	0	1
Set	DCgeneral	0	0	0	1

Table 4: Reduced graphics functions for CopyArea and friends

Again, the general function can be split into more specific derived functions for improved efficiency. Table 4 shows the values for the four parameters, and which specific function is actually used. To add planemask semantics, we force a `NoOp` in planes with a 0 by computing

```
andbits1 = andbits1 AND planemask
xorbits1 = xorbits1 OR (NOT planemask)
andbits2 = andbits2 AND planemask
xorbits2 = xorbits2 AND planemask
```

Table 5 shows the definitions of the copy-related specific functions, which are all of the form

```
Function(src, dst, andbits1, xorbits1, andbit2, xorbits2)
```

Copy Function	Definition
DCcopy	src
DCcopySPM	(dst AND xorbits1) XOR (src AND andbits2)
DCxor	dst XOR ((src AND andbits2) XOR xorbits2)
DCgeneral	(dst AND ((src AND andbits1) XOR xorbits1)) XOR ((src AND andbits2) XOR xorbits2)

Table 5: Definitions of the four CopyArea functions

Note that `DCcopy` cannot implement planemask semantics if any bit in the planemask is 0. In this case, we use a special routine `DCcopySPM`.

Whenever the foreground pixel, background pixel, planemask, or graphics functions changes, the server computes several parameters for the derived fill and copy functions. The server uses the graphics function and the planemask to compute `andbits1`, `xorbits1`, `andbits2`, and `xorbits2` for use in the generalized copy functions. The server then derives `fgandbits` and `fgxorbits` from the foreground pixel for use in the generalized fill functions, and likewise derives `bgandbits` and `bgxorbits` from the background pixel. These values are replicated four times to 32 bits in width, so that the server can paint the interior of spans four pixels at a time.

Each source file that uses the generalized fill function is compiled three times, and each source that uses the generalized copy function is compiled four times. The source code always ‘‘invokes’’ the generalized fill or copy function using C macros; the Makefile passes in the appropriate actual derived graphics function. If a derived function doesn’t use all of its parameters, then those unused parameters are never referenced by the code emitted by the C preprocessor. Thus, when a painting routine is compiled for functions like `DFcopy` the routine never reads the destination pixels or `andbits`.

7. X Imaging Modes

There are four fill styles in X: solid, opaque stipple, transparent stipple, and tile. This section describes these fill styles and techniques to paint them efficiently. All fill styles are described in terms of a single span, but remember that a shape really consists of an arbitrary number of spans. The word ‘‘merged’’ is used as shorthand for ‘‘the graphics function is applied to the source pixel and the destination pixel, and the result is written into the destination pixel using planemask semantics.’’

7.1. Solid fill

A solid fill style is the simplest possible. The server merges the foreground pixel into the destination pixels using the specified graphics function. Figure 3 shows the first three pixels of a solid fill.

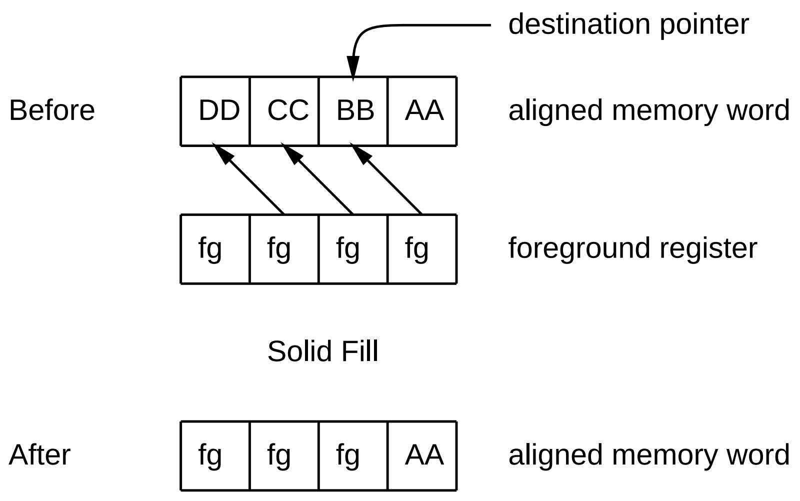


Figure 3: Solid fill style

The server uses 32-bit writes (and reads, if the graphics function requires them) to fill the interior of the span at maximum bandwidth. However, spans are aligned to pixel boundaries, not word boundaries, so the server must treat the left and right edge of a span specially. These ‘‘ragged’’ edges may not fill an entire word: at the left edge of a span the server might have to paint one to three bytes into the highest bytes of a word; at the right edge it might have to paint one to three bytes into the lowest bytes of a word.

On most architectures, the server could paint the ragged left edge in one of two ways. The server could use the bottom two bits of the destination address to perform shifting and masking operations, or to branch into four separate pieces of code. At the right edge, the server could use the same technique, but use the bottom two bits of the number of pixels still left to paint.

On the MIPS architecture, the partial word instructions `Load Word Left`, `Load Word Right`, `Store Word Left`, and `Store Word Right`—which were originally intended for accessing unaligned 32-bit words—prove unexpectedly useful for painting ragged edges.

To paint the ragged left edge, the server reads the destination using `lwr` (if necessary), combines this data with `fgandbits` and `fgxorbits`, writes the result back out using `swr`, then increments the destination pointer and decrements the width count by the number of bytes written. Since `lwr` and `swr` read and write one to four bytes, the server uses this idiom even when a pointer is already word-aligned. (If a span is so short that it is completely contained within a single word, then the server usually just uses byte accesses to paint the span.)

Painting the ragged bytes at the right edge of a span is similar. The server exits the word painting loop when there are still one to four bytes remaining to paint. The server then uses `lwl` (if necessary) and `swl` with the address of the last byte to be painted. Again, since these instructions access one to four bytes, the server unconditionally uses them at the right edge of a span.

Figure 4 shows a prototype of the solid fill painting code. This fragment is used only if the span is wide enough to touch two words in the frame buffer.

```
Pixel8    *pdst;
int       width, raggedLeftBytes;
Pixel32   fgandbits, fgxorbits;

/* Paint ragged left edge */
CFBFILLLEFT(pdst, fgandbits, fgxorbits);
raggedLeftBytes = 4 - (pdst & 3);
width -= raggedLeftBytes;
pdst += raggedLeftBytes;

/* Paint full words until 1-4 pixels left */
while (width > 4) do {
    CFBFILL((Pixel32 *)pdst, fgandbits, fgxorbits);
    width -= 4;
    pdst += 4;
}

/* Paint ragged right edge */
CFBFILLRIGHT(pdst+width-1, fgandbits, fgxorbits);
```

Figure 4: Solid fill painting code

7.2. Opaque stipple

Opaque stipple maps a bitmap into pixels. For each 0 in the bitmap the server merges the background pixel into the corresponding destination pixel; for each 1 it merges the foreground pixel into the destination. Figure 5 shows the first three pixels of an opaque stipple fill.

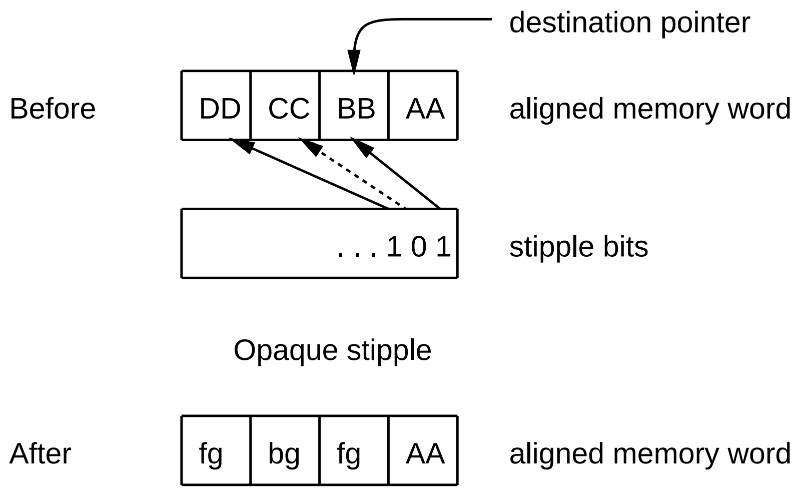


Figure 5: Opaque stipple fill style

As with solid fill, the server paints the interior of a span four pixels at a time. It maps each four bits of the stipple into the two 32-bit parameters `andbits` and `xorbits` using a 16-entry table of (`andbits`, `xorbits`) pairs. These pairs are constructed from `fgandbits`, `fgxorbits`, `bgandbits`, and `bgxorbits`, as shown in table 6; ‘‘`bga`’’ stands for 8 bits of `bgandbits`, etc. Each 32-bit entry is the concatenation of four pixels from the appropriate background or foreground bits.

Glyph bits	andbits	xorbits
0000	bga bga bga bga	bgx bgx bgx bgx
0001	bga bga bga fga	bgx bgx bgx fgx
0010	bga bga fga bga	bgx bgx fgx bgx
. . .	. . .	. . .
1110	fga fga fga bga	fgx fgx fgx bgx
1111	fga fga fga fga	fgx fgx fgx fgx

Table 6: Foreground/background mapping table for opaque stipple

When the server has at least 32 bits of stipple data to paint, it unrolls the loop 8 times. This brings painting time down to about 5.5 instructions for four pixels with the `DFCopy` graphics function, which includes loop overhead but not the instructions that may be need to fetch the stipple bits. (Remember that graphics functions like `DFCopy` don’t use `andbits` and only fetch `xorbits` from the table.)

When applications fill rectangles and other objects using stipples, they often use patterns that are of nice sizes, for which the server has special painting routines. In particular, the server paints stipples that are 32 bits wide extremely efficiently. (The server replicates stipples that are 2, 4, 8, and 16 bits wide in order to use the 32-bit painting code.) When the server paints a span, it rotates the stipple bits appropriately at the beginning of the span, then reuses these bits every 32 pixels.

With this optimization, the DECstation 3100 saturates memory bandwidth but the DECstation 5000 does not. If a span is wide enough, it is faster to look up `andbits` and `xorbits` once for each four pixels, then skip across the span painting this data every 32 bytes. On the DECstation 5000, the breakover point occurs when the span is long enough to paint each group of four pixels at least three times.

At the ragged left edge, the server uses `Store Word Right`. At the right edge, it shifts data to the most significant bytes of the source register before using `Store Word Left`.

7.3. Transparent stipple

Transparent stipple maps a bitmap into pixels. For each 0 in the bitmap the server leaves the corresponding destination pixel unmodified; for each 1 it merges the foreground pixel into the corresponding destination pixel. Figure 6 shows the first three pixels of a transparent stipple fill.

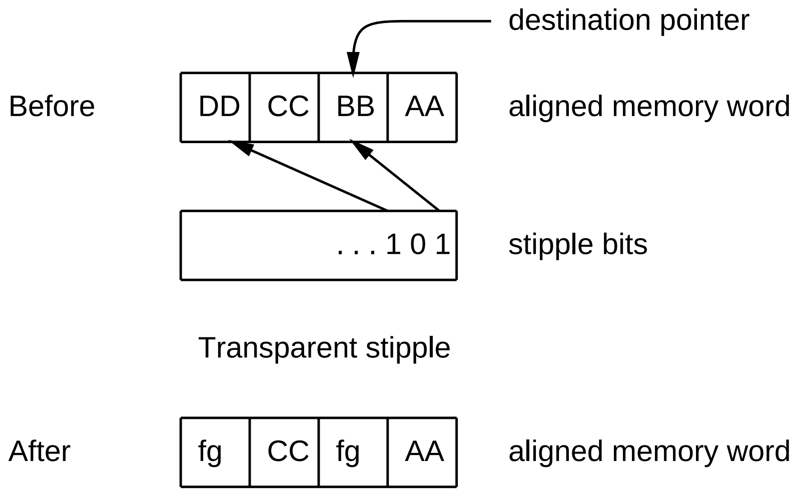
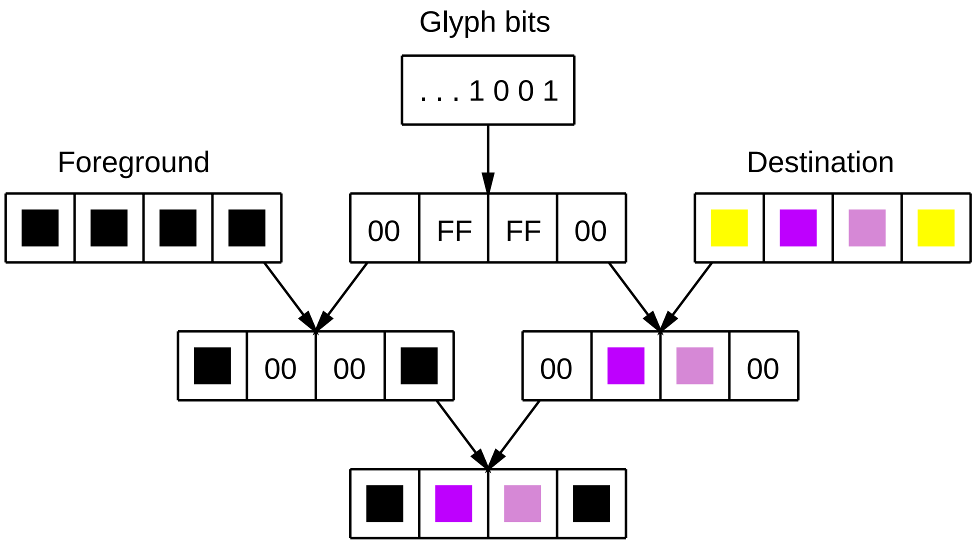


Figure 6: Transparent stipple fill style

Again, we'll focus on ways to paint the middle words of a span. The traditional implementation of transparent stipple reads the destination word, masks out the pixels to be overwritten, merges in the new pixels, and finally writes the modified word back out to memory. Figure 7 shows this process graphically, as well as the C code to implement it.

The main problem with this technique is performance. On the DECstation 3100, the best possible unrolled code takes nearly 16 cycles for each 4 stipple bits. The numbers are even worse on the DECstation 5000 due to the relatively poor read times. This technique must also be adapted to deal with graphics functions that depend upon the existing destination pixels, rather than merely overwrite them. The server cannot simply zero out the destination pixels and merge in the foreground pixels, but must perform equivalent operations appropriate to the graphics function. For example, `Xor` shouldn't mask anything from the destination, while `Invert` must invert the appropriate destination bits and ignore the foreground completely.

Instead, the four graphics functions that overwrite the destination pixel avoid reads entirely. By using the store instructions judiciously, the server can write almost any group of one to four contiguous bytes of a word in one memory cycle, as shown in figure 8. (The fg register has the foreground pixel in all four bytes.)



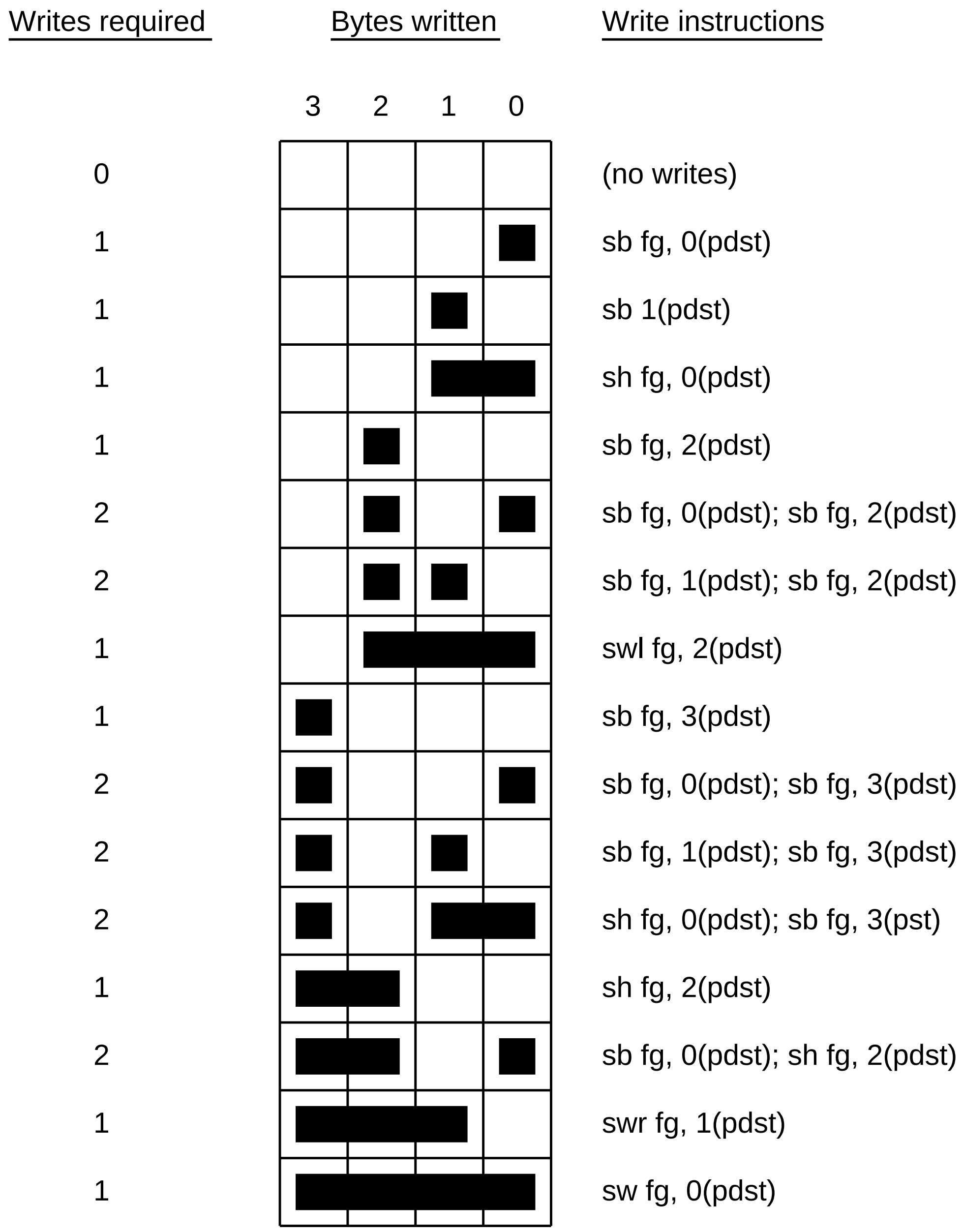
```
index   = srcbits & 0xf;       /* Get 4 bits of stipple */
mask    = masktable[index];    /* Get mask              */
dst     = *pdst & mask;        /* Mask destination      */
src     = foreground & ~mask;  /* Mask foreground       */
*pdst   = dst | src;           /* Merge the two         */
srcbits = srcbits >> 4;        /* Next 4 stipple bits   */
```

Figure 7: Transparent stipple implemented with read/modify/write

There are a total of 21 write instructions for the 16 patterns represented by four bits. If each case is equally likely, each four bits require an average of 1 5/16 writes. This is 8.75 cycles on the DECstation 3100, and 5.9 cycles on the DECstation 5000/200 (assuming page-mode writes). Since most stipples have contiguous groups of bits more often than random chance, and painting stops as soon as all remaining bits in a word are zero, we expect fewer writes on average. But even the worst case of two writes for each four bits is better than the read/modify/write strategy.

How quickly can the processor examine four bits of the stipple and branch to the right sequence of writes? The MIPS C compiler is quite good overall, but falls apart in this case. The C front end generates excessive code to dispatch a `switch`. The instruction scheduler does not move code up through the case branches in order to fill load-delay and jump-delay slots, nor does it copy tiny (one or two instruction) basic blocks up to blocks that unconditionally jump to them. The resulting code has tests that needn't be executed, unfilled delay slots, and a high branching frequency. The compiler generates 16 instructions for branch and loop overhead; adding in our average of 1 5/16 write instructions, this totals 17 5/16 instructions.

The color server uses assembly code in a few select places to implement the above technique. Coding in assembler allows optimizations that even a very good compiler and code scheduler would probably miss. In the assembly language version, all 16 case branches are exactly 8 instructions long. Rather than using a case branch table, the assembly code computes an offset from a known instruction, then branches directly to the computed location. Each case paints the appropriate bytes, then executes loop termination code for all loops surrounding the painting code. And in the assembly version, all delay slots are filled with useful work.



Writes required	Bytes written 3	2	1	0	Write instructions
0					(no writes)
1				■	sb fg, 0(pdst)
1			■		sb 1(pdst)
1			■	■	sh fg, 0(pdst)
1		■			sb fg, 2(pdst)
2		■		■	sb fg, 0(pdst); sb fg, 2(pdst)
2		■	■		sb fg, 1(pdst); sb fg, 2(pdst)
1		■	■	■	swl fg, 2(pdst)
1	■				sb fg, 3(pdst)
2	■			■	sb fg, 0(pdst); sb fg, 3(pdst)
2	■		■		sb fg, 1(pdst); sb fg, 3(pdst)
2	■		■	■	sh fg, 0(pdst); sb fg, 3(pst)
1	■	■			sh fg, 2(pdst)
2	■	■		■	sb fg, 0(pdst); sh fg, 2(pdst)
1	■	■	■		swr fg, 1(pdst)
1	■	■	■	■	sw fg, 0(pdst)

Figure 8: Transparent stipple using only write instructions

The assembly code paints four stipple bits, including loop overhead, in an average of 8 5/16 cycles. This is slightly below the (pessimistic) average memory write time of 8.75 cycles on the DECstation 3100, and somewhat larger than the write time of 5.9 cycles on the DECstation 5000. In both cases assembly language nearly doubles the painting rate for transparent stipples.

As with opaque stipples, the server uses special routines for transparent stipples that are, or can be replicated to, 32 bits wide. Unlike opaque stipples, transparent stipple code to paint four pixels is quite large, so the server doesn't unroll the loop that cycles through all 32 bits. But the server does invert the loop logic for spans wider than 96 pixels: for each four bits of the stipple, the server paints the appropriate data at intervals of 32 bytes.

The server exploits the ‘‘no-op’’ behavior of 0 at the left and right edges of a span. At the left edge of a span, the server always aligns the destination pointer to a word boundary by setting the bottom two bits to 00. It compensates for this in the stipple by logically shifting the stipple bits left, using the bottom two bits of the destination pointer to determine the length of the shift. If the bottom bits are 00, the server shifts the stipple left 0 bits; if the bottom bits are 11, the server shifts the stipple left 3 bits.

At the right edge, the server masks off any stipple bits past the end of the span. These adjustments to the stipple data effectively make all transparent stipple spans word-aligned at both the left and right edge.

The server also exploits the no-op property of 0 to stop painting a 32-pixel segment of a span when the remaining stipple bits are 0.

If the server is painting a transparent stipple with a graphics function that depends upon the destination pixels, then a 16-way branch doesn’t work well unless the stipple pattern is very sparse. The fast write instructions become slow read/modify/write sequences. Fortunately, the generalized graphics functions adapt well to this situation. The server uses a 16-entry table similar to that constructed for opaque stipples, but instead of using entries that paint the background pixel where the stipple is 0, the server uses entries that implements `NoOp` (`andbits` all 1, `xorbits` all 0). The resulting code looks much like opaque stipple code, but the server still uses the no-op properties of 0 for alignment and for loop termination.

7.4. Tile

The tile fill style copies pixels from a tile to the destination. Source and destination can be in any alignment with respect to one another. Figure 9 shows the first two pixels of a tile fill.

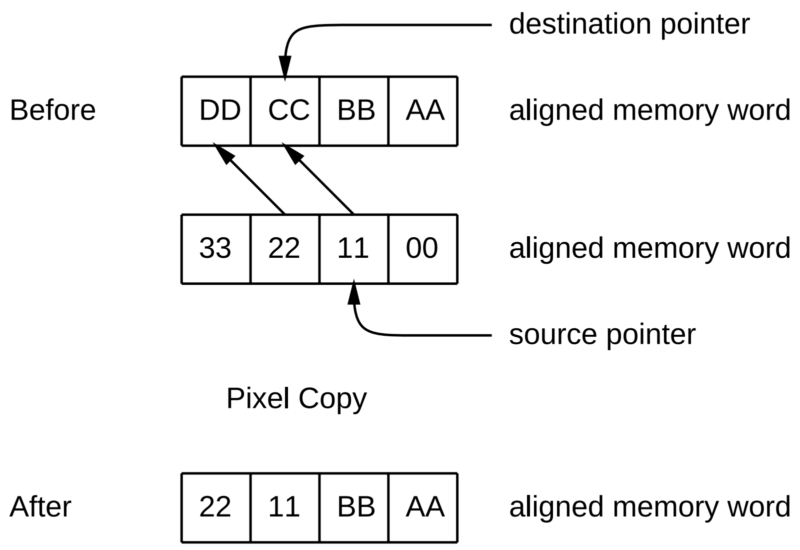


Figure 9: Tile fill style

Again, the server fills the interior of the span four pixels at a time. This is easy when the source and destination are aligned (the bottom two bits of the source and destination address are identical). The MIPS instruction set should make the misaligned case easy as well. Ideally, the server would use `Load Word Left` and `Load Word Right` to read each word of the tile. Since tiles are always stored in main memory, which is cached, this adds a single cycle per word over the aligned case. Unfortunately, it is impossible to gain direct access to these instructions from C, so the server uses the slower `CopyArea` techniques described later.

The server optimizes tiles that are 32 bits in width. This is only four pixels wide, but this is often sufficient, as each pixel may be any color. For example, the Display PostScript System[3] [7, 1] uses a tile four pixels wide by six pixels high for color half-toning. As with stipples, the server rotates the tile bits appropriately. Since the generalized copy functions are more complicated than the fill functions, the server also preprocesses the 32 bits of pixel data into the corresponding `andbits` and `xorbits`, then uses a fill function.

8. X Graphics Requests

The X protocol contains several types of drawing requests. This paper discusses a handful of them to illustrate how the techniques for various fill styles can be modified to better suit particular painting operations. `PolyText` and `ImageText` show how to modify transparent and opaque stipple painting for bitmaps that are narrow and non-repeating. `CopyArea` shows techniques for pixel copies where the source may not reside in cached memory. Finally, the section on line drawing shows how published algorithms are often expressed in a form that is non-optimal, and how these algorithms can be rewritten into more efficient forms using simple mathematical transformations.

All performance numbers were gathered by running the `x11perf` server evaluation program, with the server and `x11perf` running on the same machine and communicating via UNIX[4] sockets. The `x11perf` program measures elapsed time for the server to execute various graphics and window management requests; we used it extensively in order to time and tune our code.

8.1. PolyText

The `PolyText` request paints a string of characters onto the screen. The server first looks up the font-specific glyph (bitmap picture) for each character. The server then paints one glyph at a time using transparent stipple semantics: it paints the foreground pixel where there is a corresponding 1 in the glyph, and leaves the destination unmodified where there is a 0 in the glyph. Figure 10 shows how the character code for ‘‘Z’’ is painted.

`PolyText` is most often used by WYSIWYG editors, which allow the user to mix fonts of different heights on the same line. These editors paint a line of characters by painting a background rectangle as high as the tallest font in the line, then stencilling characters onto the background with `PolyText`.

[3]Display PostScript System is a trademark of Adobe Systems, Inc.

[4]UNIX is a trademark of AT&T Bell Laboratories.

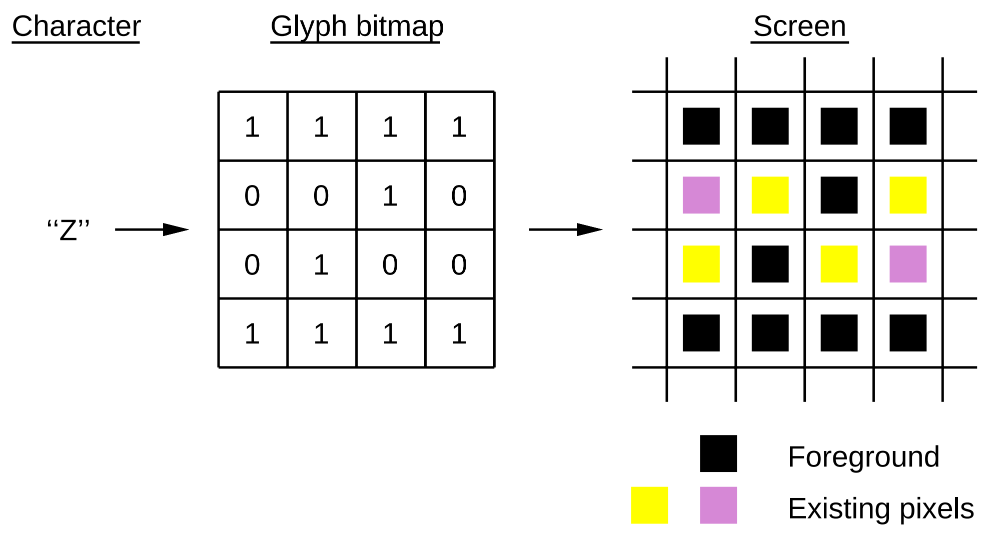


Figure 10: PolyText paints a ‘‘Z’’

Using the techniques described above for transparent stipple, the C implementation of `PolyText` paints the variable-pitch font Times-Roman[5] (10-point, 75 dots per inch) at 31,000 characters per second on the DECstation 3100, and 57,000 characters per second on the DECstation 5000. It paints the fixed width and height font 6x13 at 38,000 characters per second on the DECstation 3100, and 46,000 characters per second on the DECstation 5000.

These are respectable numbers, but don’t approach what is possible. Since glyphs are quite small, the compiler spends a relatively large amount of time fetching the glyph bits. On the DECstation 5000 the server crosses video RAM pages frequently, so many of the writes can’t use a short page-mode cycle. And as noted before, the compiler generates poor code for the case branching.

The color server uses assembly code to implement the above algorithm. The assembly code paints Times-Roman (10-point, 75 dpi) at 64,000 characters per second on the DECstation 3100, and 91,000 characters per second on the DECstation 5000. With one more improvement described in the section below, the assembly code paints the 6x13 font at 69,000 characters per second on the DECstation 3100, and 102,000 characters per second on the DECstation 5000.

8.2. ImageText

The `ImageText` request is similar to `PolyText`, but paints each glyph using opaque stipple semantics. Since different glyphs in the font may have different heights and widths—the character ‘‘:’’ might map to a 2x7 glyph, while the character ‘‘H’’ might map to a 5x9 glyph—each glyph is conceptually extended up and down with 0 bits to the overall font height. The space between glyphs is also conceptually filled with 0 bits.

`ImageText` is intended mainly for terminal emulators, which use ‘‘fixed-metric’’ fonts. Every glyph in such a font has the same width and height (for example, every glyph in 6x13 is 6

[5]Times-Roman is a trademark of Linotype.

bits wide by 13 bits high). Some text windows that allow only one (possibly variable-pitch) font also use `ImageText`, since it automatically extends all glyphs in the font to the same height.

The easiest way to implement `ImageText` is to clear a background rectangle, then use the `PolyText` code to paint in the foreground pixels. This simple technique works surprisingly well, and the color server paints all variable-pitch fonts this way. Trying to paint background and foreground simultaneously in these fonts is hard: each glyph must be extended up and down, the space between glyphs must be filled in, and in some fonts information from two adjacent glyphs can overlap (as with an overstrike character). Rather than deal with these complications, we ultimately settled for an `ImageText` rate of 46,000 (DECstation 3100) and 71,000 (DECstation 5000) characters per second for the Times-Roman font (10-point, 75 dpi).

The server painted the 6x13 font at 35,000 characters per second using this technique on the DECstation 3100. We thought we could get better performance using the opaque stipple painting techniques. Since fixed-metric fonts contain glyphs that are all the same size and never overlap, it is easy to paint the background and foreground simultaneously.

The results were disappointing. Even when coded in assembly language to get direct access to `swl` and `swr`, the opaque stipple technique improved painting of 6x13 by 3%. The problem lies in the narrowness of the glyphs. The `PolyText` technique clears a large rectangle, and deals with alignment effects only at the left and right edges. It then dabs the foreground onto this area by painting a few pixels here and there. The `ImageText` code deals with alignment on every glyph; averaging over the four possible alignments of a 6-bit wide glyph yields 1.5 writes per four bits of glyph. And overhead is quite high, as the inner ‘‘loop’’ executes at most once.

Several people had suggested that text painting be done a scanline at a time, rather than glyph by glyph. That is, the server should paint the entire top scanline of a string, then the next scanline, etc. Sketch implementations suggested that this would shuffle overhead around without improving painting rates, as the write-through cache can’t be used to assemble data efficiently.

However, this technique works well on a smaller scale, using a register to hold one row of information from a small number of glyphs. The improved `ImageText` code processed four glyphs at a time for fixed-metric fonts of width 8 or less. All four glyph rows fit into a single 32-bit word, and assembling the word is easy. This code writes partial words only at the beginning and end of each group of four glyphs, and the painting loop is unrolled. This technique painted 6x13 `ImageText` at 60,000 characters per second on the DECstation 3100, or 4.7 megabytes per second. This technique also resulted in the final performance numbers described for `PolyText` above.

Keith Packard improved this technique with a slight variation. He had no easy access to the special MIPS instructions from C, so dealing with the ragged edges at the left and right edge of a group was more difficult. His code avoids *all* partial word writes except at the beginning and end of the text string. This means reading an extra byte of glyph information to assemble each 32-bit word, but his C code is marginally faster than the above assembly code for 6x13; avoiding the extra write every sixth word pays off. Moral: don’t descend to assembly code until you’ve exhausted every other possibility.

We also noted that fixed-metric glyphs often have rows that contain only 0 bits, as each glyph has been expanded with 0 bits to fit the fixed character size of the font. Even when multiple glyphs rows are combined, there are still a lot of 0 rows. The server detects these rows and immediately generates the appropriate number of stores of the background pixel.

Incorporating all these techniques into the assembly code resulted in 71,000 characters per second on the DECstation 3100, and 110,000 characters per second on the DECstation 5000. Assembly yields only modest improvement over the corresponding C code, but most of the code was already there, so we kept it.

The server now uses this technique to paint five glyphs at a time for fixed-metric fonts of width 4 to 6, three glyphs at a time for fonts of width 9 or 10, and two glyphs at a time for fonts of width 11 to 16. This technique might profitably be extended to variable-pitch fonts if each glyph in a font were padded out to the same height. Combining glyphs into a group would be more difficult, but the ability to paint more bits at a time should more than compensate for the extra work.

8.3. CopyArea

The `CopyArea` request copies pixels from one rectangle to another rectangle of the same size. If the origins of the source and destination rectangles are assumed to be random, they are aligned (have the same two low-order bits) 1/4 of the time. In reality, many applications scroll data vertically, so the aligned case occurs much more frequently.

It would seem an easy matter to optimize the scrolling case, but experience proved otherwise. The server does some extra work at the left and right edges, but copies all words in the middle directly from source to destination, with no data massaging necessary. The hardware designers claimed that turning around the memory controller chip from read to write mode took two cycles; in order to reduce this cost, the server read in four words from the source, then wrote four words to the destination. This code should have achieved just over 5 megabytes per second, but we measured only 4.7 megabytes per second.

Some time later, we measured performance of the X11 Release 3 color frame buffer code from MIT, which scrolled large areas at 4.95 megabytes per second. We examined our code, and discovered that the instruction scheduler had pessimized our code: it had moved all loop overhead code before the stores, so that it no longer executed in the write buffer's shadow. But rescheduling the code by hand didn't improve performance.

We examined the MIT code, which copies four words at a time in an unwound loop, but always writes each word as soon as it reads it. We tried an even simpler loop, which copies a single word per iteration, and achieved a scrolling rate of 5.3 megabytes per second.

No one has adequately explained this oddity. The fastest memory writing loop takes about 400-410 nanoseconds per word. The fastest memory reading loop is about 10 nanoseconds slower. The fastest read/write loop takes a mere 760 nanoseconds per word. The moral: 2 + 2 does not always equal 4, and tuning based on measured performance may get better results than tuning based on theoretical calculations.

For unaligned source and destination pointers, the tiling technique described earlier would often be inefficient, even if written in assembly language to get direct access to `lwl/lwr`. If the source data resides on the screen, then it is uncached, and the `lwl/lwr` pair becomes quite expensive. Instead, the server reads each source word exactly once, and writes each destination word once, which requires the processor to rotate and merge words. On the MIPS, this operation takes three instructions: shift one data word left, shift the other data word right, and merge the shifted words. The server uses logical shifts that bring in 0 bits in order to avoid masking operations. (Some RISC machines, such as the WRL Titan [9], offer a register extract instruction. This instruction gets a 32-bit word from any position in the concatenation of two registers, allowing the rotate and merge operation to execute in one cycle.)

Loop overhead plus merging words is slower than the available memory bandwidth, so the server unrolls the loop eight times. When source and destination are evenly distributed among all four alignments, this code copies large areas at 4.9 megabytes per second.

On the DECstation 5000, aligned copies switch between reading and writing as infrequently as possible to take advantage of the faster page-mode accesses. For very wide copies, the server reads and writes as many as 64 pixels at a time, and achieves a scrolling rate of nearly 6 megabytes per second. Unaligned copies use the same code as the DECstation 3100, and achieve 5.1 megabytes per second.

8.4. Lines

The X protocol precisely specifies which pixels are affected for a line of width of 1 or greater. The protocol attaches looser constraints to a line of width 0; servers paint a line approximately one pixel in width, but are free to use fast software algorithms or existing graphics hardware. This section deals only with painting 0-width lines.

Bresenham's line drawing algorithm [4] uses integer arithmetic to keep track of a line's distance from the nearest pixel, and paints one pixel in each iteration. For a line with a slope between 0 and 1, the algorithm always steps right one pixel. If the vertical distance to the line from this new position is less than 1/2 pixel, the algorithm stays on the same scanline; otherwise, it steps up one pixel as well.

In the literature, Bresenham code for lines is almost always non-optimal for a software implementation. For example, References [5], [8], and [10] show algorithms akin to the following:

```
*pdst = fg;
if (e > 0) {
    pdst = pdst + scanlineWidth;
    e = e + e2;
} else {
    e = e + e1;
}
pdst = pdst + 1;
```

The variable `e` represents the distance of the pixel from the line. The adjustment `e1` represents the incremental distance to add for a horizontal step, while `e2` is the incremental distance to add for a diagonal step.

The derivation of integer algorithms for lines and conic sections in Reference [10] have a clear advantage over the other works. The resulting algorithms show how the derivations were achieved, and keep terms broken into pieces that are easy to rearrange into more efficient forms.

For example, the `else` part can be removed entirely by unconditionally adding `e1` to `e`, and compensating for this by using `e2′ = e2 - e1`. Though not an issue on RISC machines, the `if` usually generates a test instruction to set condition codes on CISC machines. This test is unnecessary if the new value of `e` is computed immediately before the test. We can move the computation if we offset `e′ = e - e1`. The improved algorithm is:

```
*pdst = fg;
e′ = e′ + e1;
if (e′ > 0) {
    e′ = e′ + e2′;
    pdst = pdst + scanlineWidth;
}
pdst = pdst + 1;
```

The color server unrolls the loop surrounding this code four times. Each point averages 5 3/4 instructions for lines with slope less than 1, and 6 3/4 instructions for lines with steeper slopes.

Early benchmarks of line drawing on the DECstation 3100 showed performance at least five times slower than projected. A program that measures handling of Translation Lookaside Buffer (TLB) traps showed that the kernel was taking 38 microseconds every time it accessed a page of frame buffer memory not currently in the TLB. On the R2000 and R3000, the TLB effectively contains 56 4-kilobyte pages. Each scanline of the color frame buffer is one kilobyte, so a page spans four scanlines. The entire screen is 216 pages, or almost four times the size of the TLB. Any line that covered much vertical distance spent most of its time faulting, not painting.

The kernel is capable of handling most TLB faults in 17 instructions, which is a tad longer than a microsecond. But the kernel had incorrectly set up the virtual memory interface to the frame buffer, which caused every frame buffer TLB fault to take a slow refill path. Once this problem was remedied, line drawing sped up about nine times, and many other operations sped up two to three times. The moral: sometimes 2 + 2 *should* equal 4, and discovering why things don’t add up may improve measured performance.

Even with fast TLB filling code, any line with a slope greater than 1 pays a substantial penalty every four pixels. While it is theoretically possible to paint long lines evenly distributed among all orientations at 2.5 megabytes per second, TLB filling reduces this rate to 2.3 megabytes per second.

The trend toward larger page sizes will reduce TLB faulting on frame buffer accesses somewhat, but frame buffers are getting larger at the same time. The best solution is to include a special TLB entry dedicated to mapping a frame buffer into virtual memory; if this entry points to a 16 megabyte page, accesses to even a very large frame buffer never cause a TLB fault.

Unconnected lines that are 10 pixels long and evenly distributed among all orientations paint at 75,000 lines per second, or about 30% of available bandwidth. If lines are connected, the DECstation 3100 can paint 110,000 lines per second. The DECstation 5000 paints 10-pixel unconnected lines at 125,000 per second, and connected lines at 160,000 per second.

9. Ingredients for a Fast Server on a Dumb Frame Buffer

Both frame buffer implementations perform remarkably well, even when compared against accelerators that were designed specifically for X. This section summarizes the factors that contribute to this level of performance.

A fast processor is essential. Not only does a fast processor paint pixels quickly, but it also dispatches X11 requests and computes window clipping information quickly. RISC processors appear superior to CISC processors for graphics operations due to the following advantages:

- Many graphics routines use 20 or more registers. RISC processors tend to have at least 32 registers, while CISC machines typically have only 16 registers.
- RISC processors usually start an integer multiply with one instruction, and fetch the result with another, which allows the the multiply to proceed in parallel with instruction execution. CISC machines usually specify a destination register as part of the multiply instruction, and stall unless they implement register scoreboarding.
- Many inner loops take the same number of instructions on CISC and RISC architectures, so the faster instruction execution time of RISC processors really pays off.
- RISC instructions don't vary in size, so assembly code can compute jump addresses directly rather than use branch tables.

The MIPS `swl` and `swr` instructions improve painting of small objects by avoiding branching logic or read/modify/write cycles. On a DECstation 3100, solid-filled 10x10 pixel rectangles paint 60% faster using `swl` and `swr` than the equivalent read/modify/write code. This advantage is even larger on the DECstation 5000 due to the long read cycle. A byte write instruction is essential for line drawing performance. And transparent stipples and `PolyText` make good use of every write instruction available on the MIPS.

In order to gain maximum benefit from unrolled loops, load and store instructions should have either a constant offset, or an auto-increment addressing mode. Neither of these should take longer than a simple indirect load.

Several algorithms make extensive use of shifting, so the CPU should include a barrel shifter.

A deep write buffer helps keep the processor from stalling after generating several writes to memory. This is especially important for ‘‘bursty’’ algorithms like text painting, where a large amount of data fetching and rearranging is followed by a large number of write instructions.

Ideally, the memory addressing architecture should support at least one large virtual address page. If not, the kernel should handle TLB faults quickly, and the base address of the frame buffer in virtual memory should meet whatever restrictions the kernel may impose.

Write bandwidth should be as high as possible, and the frame buffer should incorporate page-mode logic. Read bandwidth is less important for most applications, especially with a hardware planemask. Still, we would have liked faster reads on the DECstation 5000 frame buffer.

The frame buffer should incorporate a hardware planemask if at all possible. In particular, some imaging applications use a planemask with a graphics function of `Copy`. A hardware planemask avoids the read/modify/write used to implement a software planemask.

If there is room, hardware support for the 16 graphics functions is helpful. Most video RAMs support a read/write cycle that is only tens of nanoseconds longer than a write cycle, which is far better than sending data all the way back to the CPU.

Finally, a good compiler is vital. The compiler must be able to fit variables and partial expressions into registers over the most appropriate lifetime (i.e., much better than can be done with C’s register declarations), and must be able to schedule useful work into delay slots. Overall, the MIPS compiler does quite a good job, which kept use of assembly code to a minimum.

10. Frame Buffers vs. Graphics Processors

Frame buffers are simple and cheap to build, and you don’t have to get everything right to ship a machine. Each release of the DECstation 3100 color server has incorporated significant performance enhancements. Even so, some graphics requests are still slow; as we work out improved algorithms for these requests, we can offer better performance with no hardware changes. The initial port to the DECstation 5000 was trivial, and the page-mode logic increased performance substantially with no code changes. Performance increased even more when we tightened or unrolled loops in which overhead was larger than a page-mode write cycle.

Graphics accelerators take longer to build, but provide two advantages: the proximity to video memory allows higher memory bandwidths, and the separate graphics processor allows the server to overlap drawing with more mundane tasks.

Graphics processors from Sun Microsystems and Silicon Graphics[6] have memory bandwidths in the range of 80 to 110 megabytes per second, which is 8 to 11 times faster than the DECstation 3100 and 4 to 5 times faster than the DECstation 5000/200CX. High-speed graphics processors may get bandwidth in several ways: they use memory interleaving, they don’t communicate with memory over a general-purpose bus, and they often use a 64-bit (or wider) path to memory. Note that this bandwidth is very asymmetric: wide objects paint at 80 megabytes per second, but vertical lines usually paint at less than 5 megabytes per second.

The TURBOchannel I/O bus supports a minimum write cycle of 120 nanoseconds, which translates to about 30 megabytes per second in practice. Even on a faster bus that allows 80 nanosecond writes, a dumb frame buffer could get only half the write bandwidth that is obviously achievable in a graphics accelerator.

On-screen copy performance shows the same bandwidth discrepancy between frame buffers and accelerators. The Sun GX manages to scroll large areas over 6 times faster, and copy large areas 3.7 times faster. Even if read performance were improved on the frame buffer, there is still a big advantage to specialized hardware here.

Note, however, that even the DECstation 3100 can fill its screen in less than a tenth of a second, and copy it in less than a fifth of a second. High memory bandwidths are usually required by 2D double-buffered animation applications, but frame buffer bandwidths appear to be adequate for most other applications. Further, some simple logic can make frame buffer bandwidths comparable to these graphics accelerators.

---

[6]Silicon Graphics is a trademark of Silicon Graphics, Inc.

More important to most applications is the speed at which the server can paint small objects. CAD programs may paints hundreds of thousands of small overlapping lines and rectangles. A frame buffer offers no parallelism between setting up painting requests and performing the actual painting. Assuming all dots in a line are painted independently, both DECstations paint short unconnected lines at about 30% of available bandwidth, and the hardware doesn't even deliver the maximum bandwidth that the video RAM can provide. A server might paint short lines many times faster if some of this work were off-loaded to a specialized graphics processor.

Ironically, preliminary X11 performance numbers for accelerators from Silicon Graphics and Sun show that the dumb frame buffer code outperforms these implementations on *all* small objects, and only falls behind on large objects. Even the best accelerators from DEC paint 10-pixel lines less than twice as fast as the DECstation 5000 frame buffer. Short line performance is often limited by set-up time for each line, by data copying between application and server, and by context switching times; painting is just a portion of the total time.

Further, the ratio of memory access time to instruction cycle time is on the rise. The greater this ratio, the less time a frame buffer server spends setting up Bresenham parameters and performing Bresenham steps between each memory access. Even with today's processor speeds, note how much of this paper is devoted to fill styles, and how little is devoted to computing the outline of an object. Outline computation is already almost fast enough in most cases.

Specialized graphics hardware can offer better 2-dimensional performance than a frame buffer, but this advantage will soon be inconsequential for most applications. Like nuclear arsenals, at some point enough is enough. Users don't care if they can paint text at 200,000 characters per second using a graphics accelerator, but a mere 150,000 characters per second on a frame buffer: both devices paint a page of text instantaneously for all practical purposes. The frame buffer, once a poor man's graphics device, is sufficient for all but the most demanding applications.

11. Acknowledgements

Susan Angebranndt, Raymond Drewry, Phil Karlton, and Todd Newman wrote most of the X11 sample server and documented how to port it [2, 3].

Todd Newman wrote code to fill spans, copy pixels, paint window backgrounds, and draw polygons. Vasudev Bhandarkar and Rich Hyde wrote the device driver and driver interface.

Susan Angebranndt, Chris Kent, and Phil Karlton wrote the original `x11perf` program, which I later rewrote and added to. If you are writing or tuning X servers, you should get `x11perf` via anonymous ftp from `expo.lcs.mit.edu:/contrib/x11perf.tar.Z`.

Bob Alverson wrote the `thrash` program to measure TLB fault handling time.

Thanks to Keith Packard at MIT for always trying to outperform my code, and for the techniques and code I've stolen from him whenever he succeeded.

Mary Jo Doherty made useful comments on an earlier version of this paper.

12. References

[1] Adobe Systems, Inc.
PostScript Language Color Extensions
Adobe Systems, Inc., Mountain View, CA, 1988, 1989.

[2] Susan Angebranndt, Raymond Drewry, Phil Karlton, Todd Newman, Bob Scheifler, Keith Packard.
Definition of the Porting Layer for the X v11 Sample Server
X Version 11 Release 4 edition, Software Distribution Center, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[3] Susan Angebranndt, Raymond Drewry, Phil Karlton, Todd Newman, Bob Scheifler, Keith Packard.
Strategies for Porting the X v11 Sample Server
X Version 11 Release 4 edition, Software Distribution Center, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[4] J. E. Bresenham.
Algorithm for Computer Control of a Digital Plotter.
IBM Systems Journal 4(1):25-30, 1965.

[5] J.D. Foley, A. Van Dam.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley , Reading, Massachusetts, 1982.

[6] Gerry Kane.
MIPS R2000 RISC Architecture.
Prentice Hall, Englewood Cliffs, NJ, 1987.

[7] Christopher A. Kent.
XDPS: A Display PostScript System Extension for DECwindows.
Digital Technical Journal 2(3):64-73, Summer, 1990.

[8] William M. Newman, Robert F. Sproull.
Principles of Interactive Computer Graphics.
McGraw-Hill Book Company, New York, NY, 1979.

[9] Michael J. K. Nielsen.
Titan System Manual.
WRL Research Report 86/1, Digital Equipment Corporation, Western Research Laboratory, September 19, 1986.

[10] Jerry Van Aken and Mark Novak.
Curve-Drawing Algorithms for Raster Displays.
ACM Transactions on Graphics 4(2):147-169, April, 1985.

WRL Research Reports

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

Table of Contents

List of Figures

List of Tables