RESEARCH PAPER

A Study of the X2 Interpreter

Jim Andreas

November 24, 1987

Presented in Partial Fulfillment of the Requirements of the Masters of Science Degree in Computer Science.

RT.DOC

TABLE OF CONTENTS

1. Abstract 1

2. Introduction 1

3. Measurements of Interpreter Operation 2
   - 3.01 Profiling of the X2 Interpreter 3
   - 3.02 Dynamic Measurements of the Interpreter Operation 4
   - 3.02.01 Getfld and Setfld 5
   - 3.02.02 Expanded Objects 5
   - 3.02.03 New Standard Objects 6
   - 3.02.04 New Byte Objects 6
   - 3.03 Measurement of Object Lifetimes 6

4. Static Measurements of the Object Image 7

5. Experience Porting the Interpreter 9

6. Adding Profiling Capability to the Magnolia 10

7. Conclusions 11

Appendix A Data from the Profiling Run 18

References 21

A Study of the X2 Interpreter

1. Abstract

This paper presents data obtained from measurements of the X2 Interpreter. The measurements were made to determine how to increase the Interpreter's efficiency. The subroutine call operation was found to consume a significant percent of the execution time.

2. Introduction

The X2 software system is an experimental system developed for studying language design issues[1]. The X2 system is implemented on top of an interpreter. The interpreter is similar to a Pascal P-code interpreter or a Smalltalk interpreter. X2, like Smalltalk, uses an object-based memory that is stored in a file called an image. Transporting X2 to a different machine involves implementing the interpreter on another machine. The image needs no modification when it is transported. The overall performance of the system is greatly impacted by the speed of the interpreter.

The X2 interpreter was examined carefully by both inspecting the structure of the interpreter code and by measuring the runtime operation of the interpreter. The objective of the study was to identify the tasks which consumed the greater part of the interpreter's time. This paper reports on the results of the study. The data shows that the time spent in processing subroutine operations is very significant. The subroutine operations will be subsequently scrutinized to see how they may be modified to increase their efficiency.

The X2 interpreter was implemented initially on the Tektronix Magnolia, a 68010 based workstation with a bit-mapped monochrome display. The Magnolia implementation has been ported to several other 68000 based machines, including the Tektronix 4405 workstation. The data presented in this report was gathered from measurements made of the Magnolia and 4405 implementations.

When a procedure is compiled in X2, the code generated by the compiler is a set of bytecodes. It is the interpreter's task to execute the bytecode operations. The interpreter does this by calling routines in the interpreter which implement the operation represented by the bytecode. Many assembly language instructions are executed to interpret each bytecode.

Objects are loaded from disk files as they are needed. When an object is accessed that is part of the virtual image, but has not yet been read into memory, the interpreter automatically reads in the object into memory.

The X2 interpreter utilizes two workspaces to manage the garbage collection of objects. When one workspace is full, the objects which are still 'live' are copied to the other workspace, and the original workspace is declared empty. When the other workspace becomes full, the process is repeated[3].

I/O primitives allow functions to obtain mouse coordinates and status of the mouse buttons, as well as keyboard data.

The interpreter provides a primitive BitBlt (bit block transfer) operation to display raster images and text. The BitBlt algorithm was not included in the experiments performed on X2.

3. Measurements of Interpreter Operation

In order to better understand the operation of the X2 system several measurements were made. These measurements fall into two general categories: static and dynamic. The static measurements were made of an X2 image file. These measurements examined the object sizes and degree of connection between objects.

A number of dynamic measurements were made of the interpreter operation. Once the profiling capability had been added to the Magnolia (see Chapter 6) the X2 interpreter was measured to find out where it was spending its time. Instrumentation code was added to one version of the X2 interpreter to observe operation of some of the bytecode instructions. Some of the measurements were made to see if a change in the encoding of the bytecode operations might improve performance.

All of the dynamic measurements were made on images set up by Prof. Sandberg to run a standalone set of operations. The test run did not do any graphics output to the display or require any user interaction and so were invariant from run to run.

This command caused the x2 interpreter to load objects from the two image files, "test1" and "newbase1". The objects were loaded first from test1, and objects were read from newbase1 as required. In this example, newbase1 is the primary image file, and test1 is the secondary image. By using the two files together, the definition of the objects in the primary file can overridden by objects in the secondary file. Another component of the measurement testing was a file called "example", which contained some code to compile.

3.01 Profiling of the X2 Interpreter

The profiling facility on the Tektronix 4405 workstation was used to measure where the X2 interpreter was spending its time. The operating system profiling facility periodically interrupts the running process, examines the program counter for the process, and increments an array value that corresponds with the program counter.

The measurement was made on the 1.2 version of the X2 interpreter running on a Tektronix 4405 workstation. The workstation supported profiling, so very little extra work was necessary to modify the system or the interpreter for the profiling measurement. Early profiling work was also done on the Tektronix Magnolia, which was more difficult (as described in chapter 6).

The profiling data presented here is grouped into nine functional areas of the interpreter. During the course of repeated profiling runs and analysis, the data hinted that the subroutine operations were the functional block which used the largest percentage of the interpreter's time. To isolate the time spent in executing the subroutine operations, all common interpreter routines executed were duplicated and renamed so that a unique set of subroutines would be executed. The renamed routines could then be traced uniquely in the profiling data, and their execution percentage could be allocated to the correct functional block.

The command line used to run the measurement was:

```
x2 -p base2 test2
```

Figure 1 shows the percentage of time spent by the interpreter in the nine functional areas. Appendix A gives the raw data obtained. The data shows that the time spent executing the subroutine operations is very significant. The interpreter spent 47.5 percent of its time during the test run executing subroutine operations.

The results derived show the percentage of the total user process run time spent in the various interpreter functional blocks. These are summarized below:

Table 1 – Profiling Data

47.5%	Subroutine operations The subroutine call operation in X2 is a fairly expensive operation, and this results in a high percentage of the total interpreter time being spent in its execution. The subroutine call must allocate a new object to contain the activation record, and read fields from three objects to set up pointers before execution of the subroutine may begin.
11.0%	Interpreter loop

Each bytecode instruction is dispatched by the interpreter loop. This loop picks up the next bytecode to be executed, looks up the address from a table of the routine to execute the bytecode, and dispatches control to the routine. This routine represents the "overhead" of the interpreter. Some interpreters seek to avoid this overhead by providing for the compilation of a set of bytecodes into native code [4].

10.5% Garbage collection

An activation record object is created each time the interpreter executes a function or procedure call bytecode. The activation record object is used ("live") until the interpreter executes a return from a function or procedure primitive. The activation record object then is garbage taking up room in the workspace. Other garbage is accumulated as objects are created then discarded. When the workspace fills up a sweep is made of the workspace to evacuate objects still in use to a second workspace. The workspace is then "condemned" for reuse. The data shows that the interpreter spends a large fraction of its time collecting garbage.

6.0% Data manipulation operations

This category refers to operations such as indexing or obtaining the field of an object.

6.0% File input and output

During the profiling run X2 wrote a large number of messages to the screen. This percentage reflects the time spent writing the messages.

5.5% Push operations

Various operations push constants, variables, and pointers to other objects onto the top of the stack.

5.0% Store operations

2.5% Arithmetic operations

3.0% Control flow operations

3.0% Miscellaneous operations

3.02 Dynamic Measurements of the Interpreter Operation

The dynamic measurements were made of the interpreter with an intent to see if the interpreter's efficiency could be improved. Prior to these experiments, Prof. Sandberg had examined the dynamic frequency of the various byte code operations. His study showed that the execution of one particular byte code operation, Lval (push a local variable's value onto the stack) accounted for nearly 35 percent of the execution of all bytecodes. This implied that an improvement in performance of the Lval instruction could greatly improve the performance of the interpreter. The interpreter and X2 compiler were modified to add eight new bytecode instructions, Lval0 through Lval7, which "wired in" the offset to the variable which is to be pushed on the stack. This increased the efficiency of the interpreter by removing the time necessary to pop the offset to the variable from the stack.

The interpreter was studied further in an effort to find more useful areas to improve efficiency. The command line used to run the measurements was:

```
x2 -p newbase1 test1
```

3.02.01 Getfld and Setfld

The Getfld (Get a field of an object) and Setfld (set a field of an object) are two very important interpreter operations. Every reference to a variable is made via these operations. The operations are passed a pointer to an object, and an offset into the object representing the field to be referenced. When a standard object is referenced the offset passed is an integral number of words, while with a byte object the offset is an integral number of bytes.

The measurements indicate that the Getfld operation was executed an order of magnitude more times than the Setfld operation.

The two operations were instrumented to determine the frequency distribution of the field numbers passed. Figures 2 and 3 show the results from the profiling run. The Getfld distribution shows that the overwhelming number of accesses are made to field zero of objects. This is primarily due to the icall bytecode operation. The icall implementation performs three Getfld calls to field zero as it obtains information about the object to be called. A subsequent measurement showed the calls made by icall represented 83% of the total calls made to Getfld to obtain field zero.

3.02.02 Expanded Objects

The expand operation is responsible for reading objects from a diskfile into memory. The interpreter calls the expand routine when it detects that an object has not been read into memory. Expand performs a disk seek to the position of the object in either the primary or secondary image file. It then reads the object into the next available area in memory.

Standard object's fields are scanned after the object is read into memory. Pointers to other objects stored within the standard object are flagged. This indicates that the objects pointed to may not have been read in.

The expand operation was instrumented to record the sizes of the objects expanded from a file. The results are shown in Figure 4. The chart shows that the great majority of objects read into memory are of size 2.

3.02.03 New Standard Objects

One of the most common actions in the interpreter is the creation of new objects for use as activation records. Figure 5 shows a comparison between standard objects created for use as activation records versus for other uses. The activation records are created no smaller than size 4, as they require space for a number of standard fields. Figure 5 shows that the sizes of activation records cluster primarily from size 6 to 9. This indicates that the functions called during the test run had few local variables and required very little stack space.

The majority of the other standard objects created are of size 2. This is in part due to the convention in X2 of using a size 2 standard object to refer to the address of a field of an object. The reference is made by a standard object which contains a pointer to the object and the field's offset within the object. A number of the bytecode operations create a field reference object and return the object's address on the stack.

Table 2 – X2 Primitives which Create Objects of Size 2

Creations	%	Description
134,804	35	Created by the load addr (Iladd) primitive
108,858	28	Created for use as a thunk (Ipushn)
90,672	23	Created for general use (Icreate)
47,854	12	Push the addr of a field of an object (Isubs)
3,444	1	Push the addr of a field of an object (Iindex)
1,764	0	Created for general use (Snew)

3.02.04 New Byte Objects

Figure 6 shows the distribution of byte objects created over the course of the test run. The figure shows two peaks, one at size 1 and one at size >= 20. The peak at size one is most likely due to the creation of single character literals. The peak at >= 20 can be attributed to the manipulation of character strings or the creation of bytecode objects from the compilation of functions in the test run.

3.03 Measurement of Object Lifetimes

The garbage collection algorithm used in the interpreter is a Baker type two-space garbage collector[3]. The Baker algorithm uses two spaces in memory, space 1 and space 2. Objects which are created during the interpreter operation are allocated from space 1. When the space is filled, a scan is made of space 1, and objects still in use are copied to space 2. When the collection is complete, pointers to space 1 and 2 are swapped.

The Baker scheme is modified in X2 by the addition of a third workspace. The third space contains objects which are read from the image file upon demand. The third

space is not scanned by the interpreter's garbage collector. If an object in the third space is modified, the object is moved into space 1, and so becomes subject to garbage collection.

The X2 interpreter was modified so that object lifetimes could be measured. The algorithms which create objects were modified to include an extra field at the end of each object. The field was filled with a timestamp indicating when the object was created. The garbage collection algorithm was modified so that when objects were scavenged from a condemned workspace into new workspace, the extra field was copied also. When a garbage collection pass occurred, an analysis pass was made through the condemned workspace. All objects which were not reclaimed from the condemned workspace were logged.

It is difficult to detect precisely when an object is no longer referenced. The garbage collection algorithm can detect references, but the algorithm is very expensive in time. To enhance the accuracy of the measurement, the size of space 1 and 2 were decreased. Space 1 filled more quickly, and thus forced more frequent garbage collections.

The measurement results are shown below. The data shows that over 97 percent of the objects created were reclaimed by the next garbage collection pass or the subsequent pass (the 0 or 1 rows). Longer lived objects, surviving 6 or more garbage collection passes, amount to about 2 percent.

Table 2 – Lifetime of Objects
as a Function of Garbage Collection Passes

number of passes	number of objects	percent of total
0	1,546,619	94.89%
1	35,230	2.16%
2	7,022	0.43%
3	2,468	0.15%
4	1,582	0.10%
5	1,534	0.09%
6 – 572	35,454	2.18%

total objects logged 1629909

4. Static Measurements of the Object Image

Static measurements were made of the X2 virtual image file by developing the isize utility. The isize utility scans an image file and prints out statistics about the contents of the image. The utility reads the entire image file into memory, and then scans the image. As isize scans the image, it logs data about the objects contained in the image.

isize then prints the following information about the image:

o A count of the number of byte and standard objects.

- The total size of each type of object.
- The maximum byte and standard object sizes.

Standard objects can contain either integers or pointers to other objects. Isize logs the type of data found in standard objects. It reports on:

- The relationship between the total number of integers versus pointers to other objects.
- The proportion of pointers to byte objects versus pointers to standard objects.
- The average reference count (numbers of pointers to) byte objects versus standard objects.

In addition, if the verbose option is selected the isize utility will print a set of tables which lists the sizes of the objects in the image, and how many objects are present of each size.

Isize does not support secondary image files. The isize utility was developed prior to the support of a primary and secondary virtual image file scheme. To examine a combination of primary and secondary image files, the utility x2compress should be run to combine the primary and secondary file into one compressed primary file.

The isize utility reported the following results on the "newbase1" image file:

Table 3 – Data Reported by the Isize Utility

```
byte object count       5173 (31%)
standard object count  11460 (69%)
total objects          16633

byte object space      434776 bytes (66%)
standard object space  215196 bytes (34%)
total image size       649972 bytes

max byte object size        6104
max standard object size    3612

for standard objects:
Number of int refs: 11368 (27%)
Number of obj refs: 30971 (73%)
===========================
Standard object information:
===========================
byte refs:      5427
std obj refs:  25544
Average ref count for byte objects is:      1.05 counts
Average ref count for standard objects is:  2.23 counts
```

5. Experience Porting the Interpreter

An early version of the X2 interpreter was ported as an experiment to a proprietary HP Unicorn workstation. The Unicorn workstation is a 68000 machine which runs a version of the AT&T UNIX(tm) version 3.0 operating system. The Unicorn workstation project was canceled, however a number of prototype workstations were built. The X2 interpreter was ported to a workstation which was equipped with an 8 Mhz 68000 and a 20M byte hard disk.

THE ASSEMBLY LANGUAGE SYNTAX PROBLEM

The first problem in porting the interpreter dealt with the difference in the assembly language syntax. Most of the interpreter is written in 68000 assembly language for the Magnolia UNIX assembler. The assembly language syntax for the Magnolia is described by the following example:

```
saveimage:
     movl d3,sp@-
     jsr  swap2    | position at end of objects on file
     clrl sp@-
     movl #4,sp@-
     movl file+4,sp@-
     jsr  lseek
```

The equivalent assembly syntax for the Unicorn assembler is shown below:

```
saveimage:
     movl d3,-(sp)
     jsr  swap2    ; position at end of objects on file
     clrl -(sp)
     movl #4,-(sp)
     movl file+4,-(sp)
     jsr  _lseek
```

The assembly language conversion problem was solved by writing scripts for the UNIX stream editor SED. The SED scripts converted the majority of the assembly language syntax. A small amount of hand editing was still required to fix up underbars. The Unicorn C compiler prepended underbars to function and global data names, while the Magnolia compiler did not.

This done, the first test object was run. The test object was a small file of X2 objects which, when run by the interpreter, would loop incrementing a variable from 1 to 10, and print the value of the variable to the UNIX standard output. After some debugging of the bytecode dispatch routine, the test ran successfully.

THE DISPLAY BUFFER PROBLEM

The next problem was connecting to the display memory. The Unicorn had a display buffer which could not be written directly. Instead all accesses to display memory had to

be made through a Graphics Processing Unit (GPU) chip. The GPU synchronized accesses to display memory, placed characters into the display buffer, and did block copies of rasters to and from the display buffer.

The initial method used to hook X2 to the GPU was to use a scratchpad buffer from which X2 would write and read. This buffer was allocated within the interpreter. After every display operation the scratchpad buffer was copied through the GPU to the display buffer. After some debugging the X2 interpreter was able to come up and display its windows! However, since a full screen copy was made after each character write, the system was very slow. An optimization to block copy only the area of memory modified by the most recent interpreter display operation speeded up the display, but was still annoyingly slow.

THE HP MOUSE PROBLEM

The last problem involved the HP Mouse, and was never satisfactorily solved. The X2 interpreter is designed to use a three button mouse, and fully utilizes all three buttons. The HP Mouse is a two button mouse, but getting both mouse buttons hooked into X2 was very tricky. A special polling routine was hooked up to query the mouse when the interpreter's I/O routine used to obtain mouse coordinates was called. A function key supplied the missing mouse button. The system worked, but the interface was much clumsier than the Magnolia implementation.

A LOOK AT A DIFFERENT MACHINE

I examined the potential problems involved in porting X2 to a HP Series 300 machine. Unfortunately I ran into a snag with the display buffer layout. The display buffer for most of the current Series 300 displays is oriented toward a byte per pixel. The Magnolia and the Unicorn both implement the buffer as a bit per pixel. Converting to a byte per pixel, if done correctly, meant rewriting the BitBlt modules. This is a major task. I tried an experiment in an effort to create a less expensive solution where instead of writing a byte to the display buffer, the bits were unpacked and written to successive memory locations. Conversely, before a byte was read from the display buffer, special code was inserted to assemble the byte from eight successive display buffer locations. Unfortunately this effort never resulted in meaningful patterns on the screen, and the port was abandoned.

CONCLUSIONS

A interpreter environment which intensively uses raster graphics for its user interface requires a powerful workstation as its underpinnings for acceptable performance. If access to the display buffer is restricted by a mailbox type architecture like the GPU based implementation, performance suffers severely.

6. Adding Profiling Capability to the Magnolia

Profiling refers to a method of sampling the program counter (PC) of a processor while it runs a program. The samples are then used to determine the time spent by the processor executing various routines in the program.

There are two pieces to a profiler. The first piece is a sampling mechanism and a set of buckets. The sampling mechanism takes a snapshot of the program's PC at a regular interval. The mechanism then finds the bucket which corresponds to the PC and increments the number in the bucket.

The other piece of a profiler is a utility program to analyze the bucket data. The utility program provides a printout of the time spent in each routine in the program.

The Tektronix Magnolia was the only workstation on which X2 was running during the early measurements for this paper. Unfortunately the Magnolia did not support profiling in its operating system nor did it provide the utility to analyze the results. So the following was done:

The Magnolia's operating system had "hooks" for the sampling mechanism, but was missing a significant routine. The Magnolia's operating system was modified to add the missing routine. The routine was called every hardware clock tick to increment the bucket appropriate for the program counter location in the program being tested. Before the program exited, the operating system was queried to obtain the contents of the buckets.

Second, a version of the prof utility was ported to the Magnolia. Portions had to be rewritten for the differences in the Magnolia's symbol table. Once this was done, the bucket data could be read by prof, along with the symbol table information of the X2 interpreter. The prof utility would then print out the percentage of time spent in the various routines in the X2 interpreter.

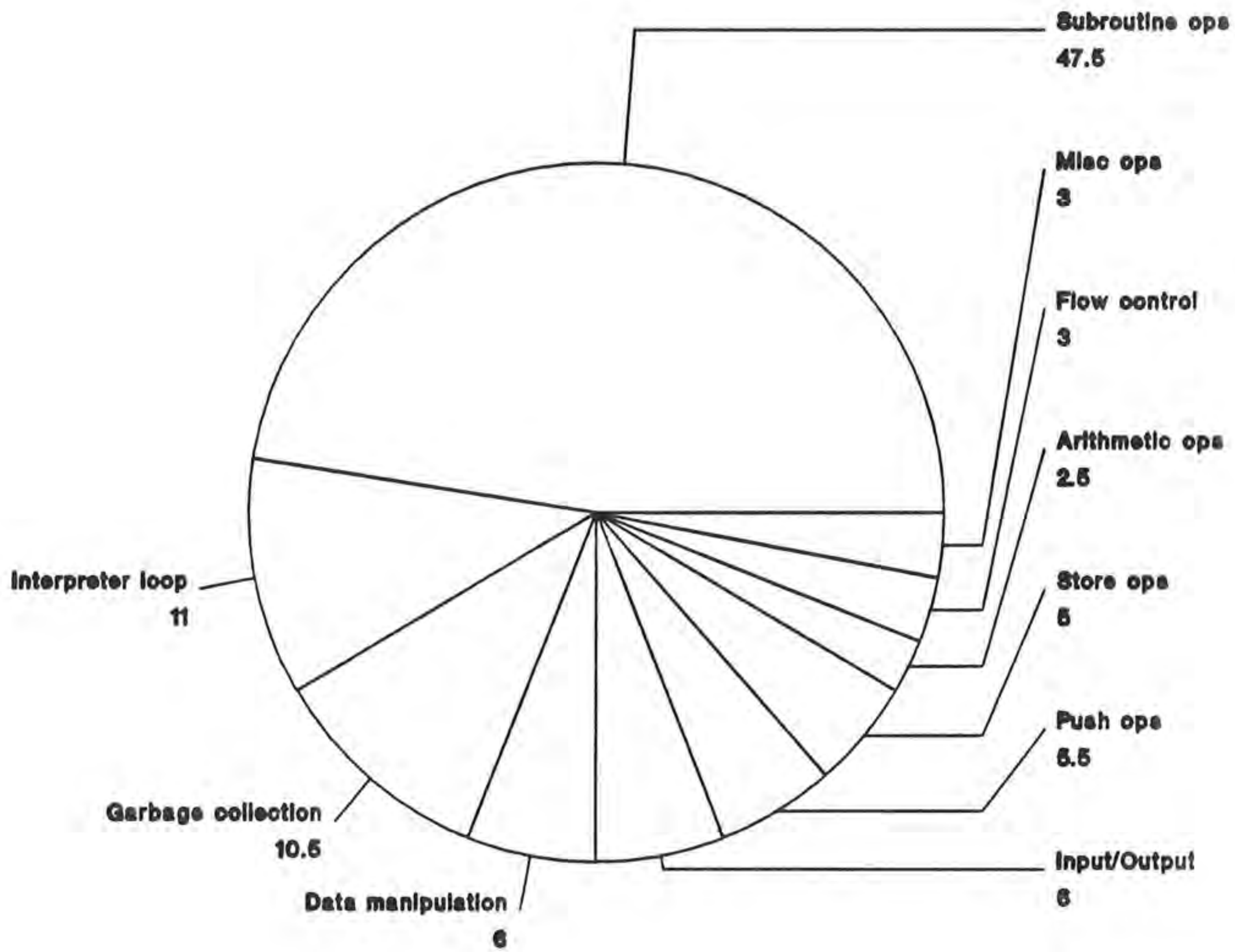
7. Conclusions

The profiling of the X2 interpreter yielded the most interesting results in this study. The profiling data presented in Chapter 3 shows that the execution of subroutine operations require nearly 50 percent of the interpreter's time. Therefore any effort to increase the interpreter's efficiency must focus on optimizing the subroutine operations.

The efforts to port the interpreter to the HP workstation yielded a good understanding of the interpreter's operation, but very little in terms of data. The work to add profiling capability to the Magnolia workstation provided early results in profiling the interpreter, but the results obtained were dwarfed by the investment of time to obtain them. Subsequent data obtained by measuring the Tektronix 4405 implementation were far more fruitful.

Figure 1:

Time Analysis of the X2 Interpreter



Values are in percent

Figure 2:

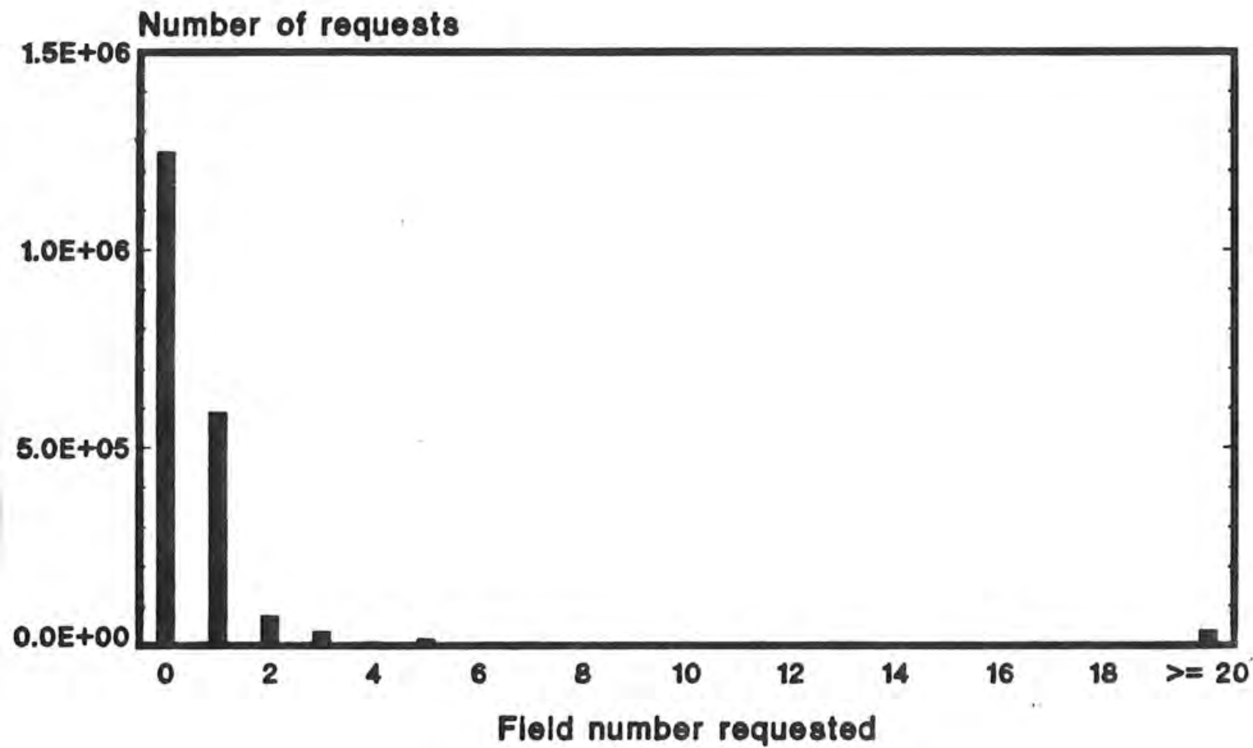


Figure 3:

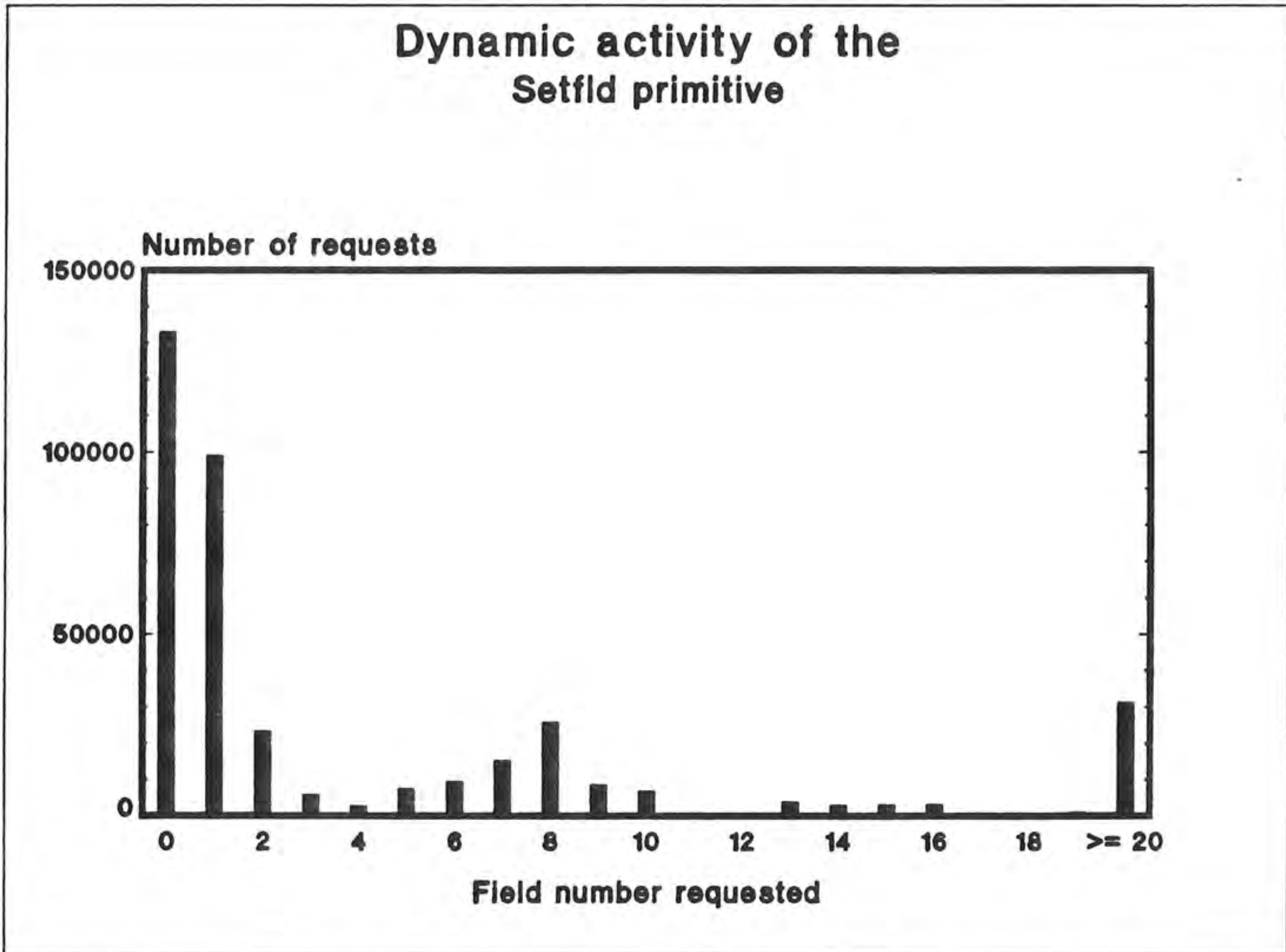


Figure 4:

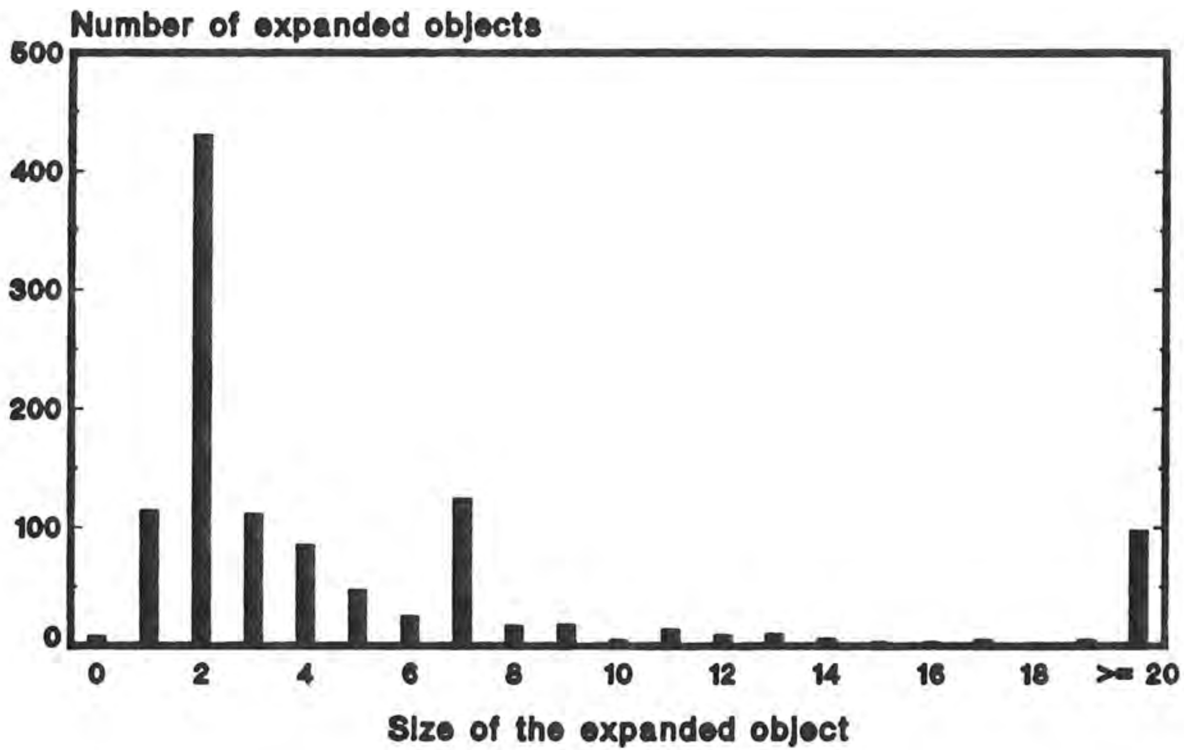


Figure 5:

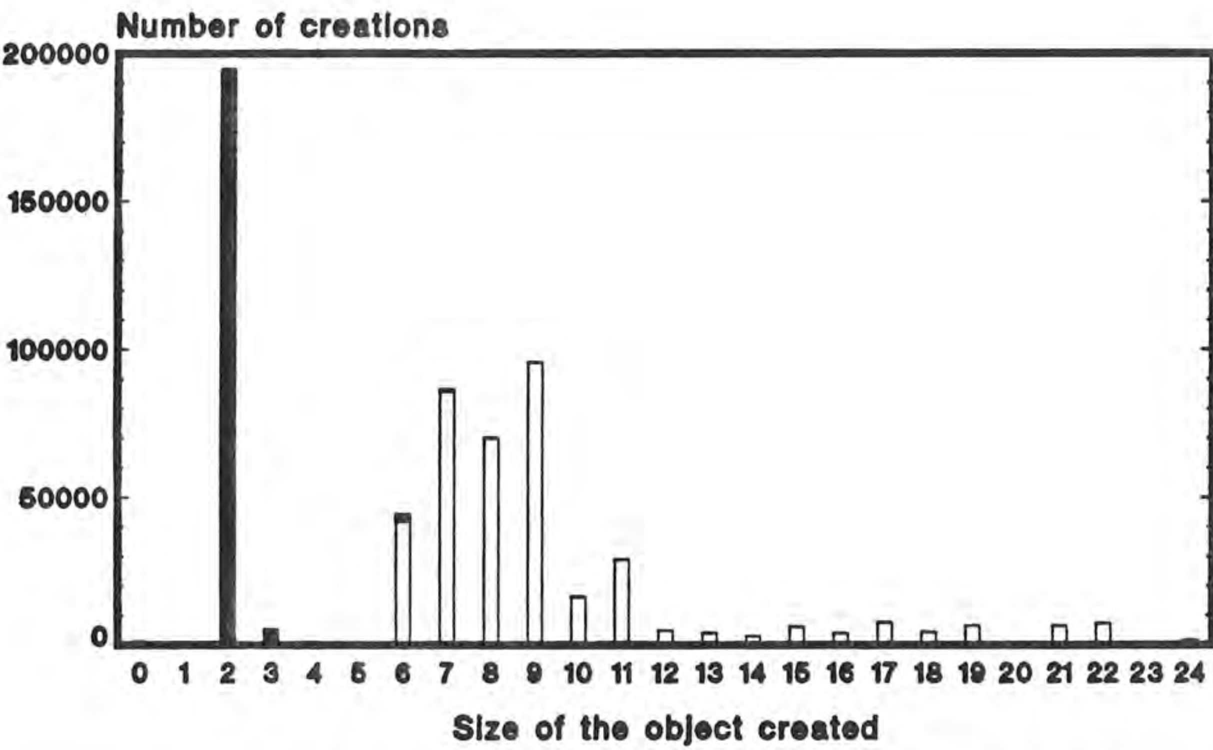
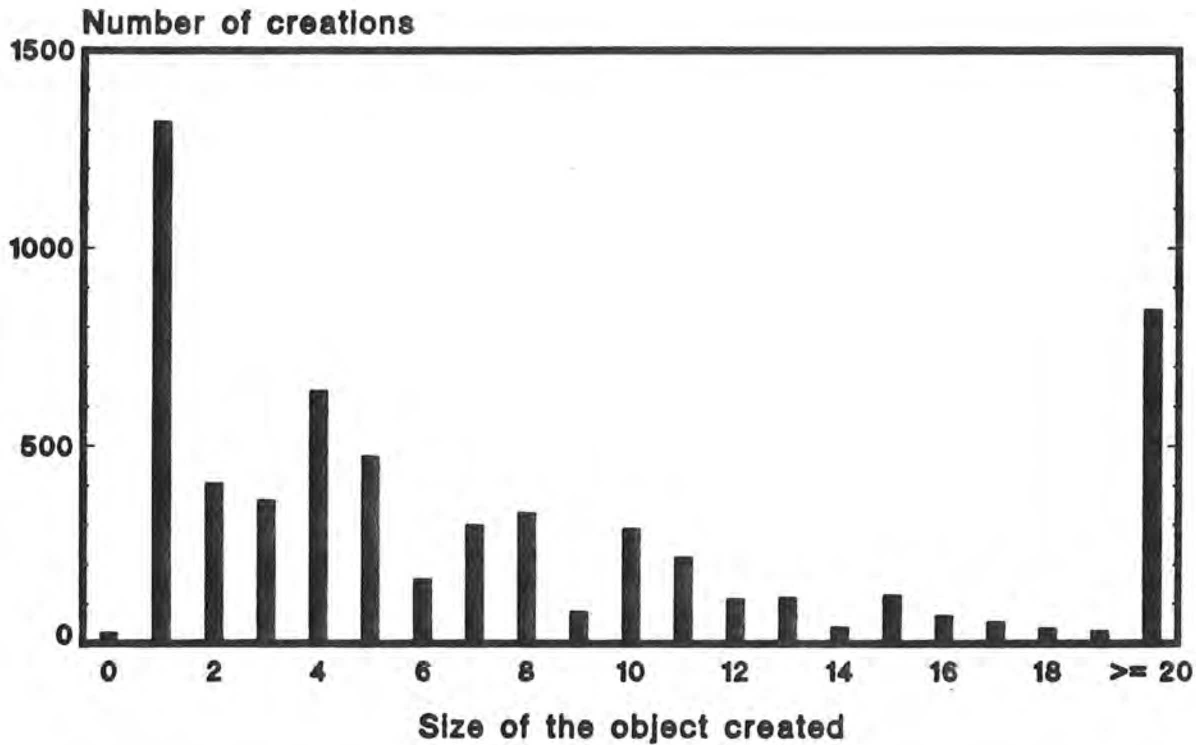


Figure 6:



Appendix A Data from the Profiling Run

This appendix shows data collected from profiling the X2 interpreter. The profiler samples the interpreter operation and reports the percentage of time spent in the various interpreter routines. The data shown below has been organized into functional groups of operations.

The profiling measurement was made on a Tektronix 4405 workstation running X2 version 1.2. The command line used for the measurement was "x2 -p base2 test2". The test1 image opened a file named "example" and compiled the code in it.

Many profiling runs were made of the X2 interpreter. The interpreter was modified in some areas to duplicate common subroutines so that a clear allocation of time could be determined for some operations. The symbols with the prefix "test" in the name are an example of this. For example, it was theorized that the subroutine operations in the interpreter were taking a large fraction of time. By duplicating the many subroutines called to interpret a subroutine bytecode into unique subroutine names, it was determined that the subroutine operation interpretation was indeed using a significant percentage of the interpreter's time (47%).

```
Iret             0.1
Iretv           1.01
Igcall          0.16
Icall           7.18
Ipushn          1.23
testrestore     0.85
testloc7        3.75
testloc8        1.91
testloc9         1.4
testgetfld2     0.24
testgetfld     10.37
testnewobj      1.08
testlocnew      1.87
testgetadd2     6.44
testsgetadd     9.81
     Total:    47.40     Subroutine operations

_runobj        11.06
     Total:    11.06     Interpreter loop

swap           10.51
     Total:    10.51     Garbage collection
```

Dnewobj	0.01	
Dlocnew	0.13	
Dgetfld2	0.59	
Dgetfld	3.37	
Idref	0.58	
Iindex	1.45	
Total:	6.13	Data manipulation operations
expand	0.01	
addentry	0.05	
_lseek	0.05	
_read	0.47	
_write	5.35	
Total:	5.93	File input/output
Pnewobj	0.23	
Plocnew	0.16	
Ilval	3.42	
Iladd	0.32	
Iobref	0.46	
Iconst	0.89	
Total:	5.48	Push operations
Ipopl	0.77	
Ipopsub	1.14	
Sgetfld2	0.11	
Sgetfld	0.67	
Ssetfld	2.45	
Total:	5.14	Store operations
Ieql	0.71	
locl2	0.38	
Iadd	0.92	
Asgetadd	0.2	
Smult	0.23	
Total:	2.44	Arithmetic operations
Ijmp	2.79	
Total:	2.79	Control flow operations

loc9	0.04	
Ihalt	0.01	
Icreate	0.27	
Inot	0.08	
Itran	0.03	
Sisint	0.47	
getfld	0.69	
locget	0.14	
locget3	0.51	
locget4	0.12	
cobjsize	0.03	
objsize	0.3	
locsize	0.11	
getadd2	0.03	
isint	0.08	
newobj	0.11	
locnew	0.09	
cnewbobj	0.01	
newbobj	0.01	
Total:	3.13	Miscellaneous operations

References

1. David W. Sandberg, "Experience with an Object-oriented Virtual Machine", *Software Practice and Experience*, 1988.

2. Adele J. Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addision-Wesley, Reading, MA, 1983.

3. Baker, H.G., "List Processing in real time on a serial computer," *Communications of the ACM 21*, 4 (April 1978), pp. 280-294.

4. Mark B. Ballard, David Maier, Allen Wirfs-Brock "QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods", Proceedings OOPSLA 1986, ACM, 1986.