# VIPS: A Visual Debugger For List Structures

Takao Shimomura and Sadahiro Isoda

NTT Software Laboratories
3-8-1 Kohnan, Minato-ku, Tokyo 108 Japan

**Abstract**

In order to find errors in a program, it is very important to visualize the program execution. Various kinds of program execution visualization techniques have been developed for program debugging. However, existing techniques have some problems such as they require users to define the type of figures for each data item in advance and they cannot execute a program rapidly enough for practical use.

The VIPS debugging tool has been developed to solve these problems by automatically acquiring data type information from the program to be debugged and rapidly displaying data structures. The current version of VIPS aims at list structures which are the most difficult type of data structures for debugging. A preliminary evaluation shows that VIPS enables a user to find bugs in about thirty percent less time using about twenty-five percent less debugging commands compared with a conventional debugging tool.

## 1. Introduction

### (1) Survey of Program Execution Visualization Techniques

In order to find errors in a program, as well as to understand its process, it is very important to visualize the program execution. For the purposes of programming education and debugging, various kinds of program execution visualization techniques have been developed so far. These techniques are classified into the following two categories.

One is a multiple-view technique which makes it possible to display various aspects of program execution simultaneously. Systems classified into this group display various kinds of information using multiple windows. Dbxtool [1] displays the source text of a program being debugged and it indicates the current execution line and lines where breakpoints have been set. It also displays values for specified variables every time the execution of the program is intermitted. Ups [2] displays values of variables according to their hierarchical data structures. Mtdbx [3] displays the state of each task in a parallel processing program.

The other category is a visualization technique which enables program execution to be displayed graphically. Systems classified into this category display program control and data structures as figures in order to facilitate the understanding of the control flow and changes in data values. There are several different systems in this category. PECAN [4] shows the control flow not only in a program text but also in a program chart. BALSA [5] displays program algorithm as figures. PROVIDE [6] displays data values as boxes, bar charts or pie charts, etc. Amethyst [7] displays the call stack and data values as figures. DS-Viewer [8] displays data structures according to memory location. BALSA [5] and Amethyst [7] were developed for programming education. They are used for teaching programming algorithms like sorting, hashing and pattern-matching ( BALSA ), and programming facilities such as parameters and recursive calls ( Amethyst ).

In 1987, the authors developed the first version of the visual debugger, VIPS(V1), which can display program execution as figures. VIPS(V1) uses the multiple window mechanism to realize such facilities as: (1) displaying the control flow in both the program text and a module structure chart, (2) displaying the call stack as figures, and (3) displaying data structures as figures which represent data semantics ( such as a stack figure, or a bar chart ).

### (2) Problems of Program Execution Visualization Techniques

The existing systems mentioned above, however, have some problems: (1) they require users to define the type of figures for each data item in advance, and (2) they cannot execute a program rapidly enough for practical use. Dbxtool [1], Ups [2], mtdbx [3], and PECAN [4] do not have enough ability to display program execution as figures. PROVIDE [6] and DS-Viewer [8] can express data structures as highly abstracted figures, but they require users to define the type of figures in advance. Because VIPS(V1) [9] adopts an interpreter for executing programs, its speed is about two orders of magnitude slower compared with normal execution.

Therefore, the authors have developed VIPS(V2) to solve the problems mentioned above. It can acquire data type information necessary for automatic display of data structures as figures and rapidly display the part of data structures that need checking. The newer version of VIPS can visually display only list



0730-3157/90/0000/0530$01.00 © 1990 IEEE

structures. This is because the authors think that list structures are the most difficult type of data structures for debugging and thus they deserve early implementation.

A preliminary evaluation shows that VIPS enables a user to find bugs in about thirty percent less time using about twenty-five percent less debugging commands compared with dbxtool [1] which is one of the conventional debugging tools. As to performance, VIPS is very practical in that it can display even large list structures within one to four seconds.

The following sections show the program execution visualization facilities VIPS provides, its implementation, and the results of a preliminary evaluation.

## 2. Program Execution Visualization Facilities

### 2.1 Multiple View Facilities

VIPS visualizes the execution of programs written in C language from various aspects using multiple windows ( Fig. 1 ). The main VIPS windows are described below.

**(1) Monitor window:**

Accepts debugging commands and displays their responses. Frequently used commands are assigned buttons or menu items.

**(2) Program-text window:**

Displays the source text of the program being debugged. It indicates the current execution line and lines where breakpoints have been set.

**(3) List window:**

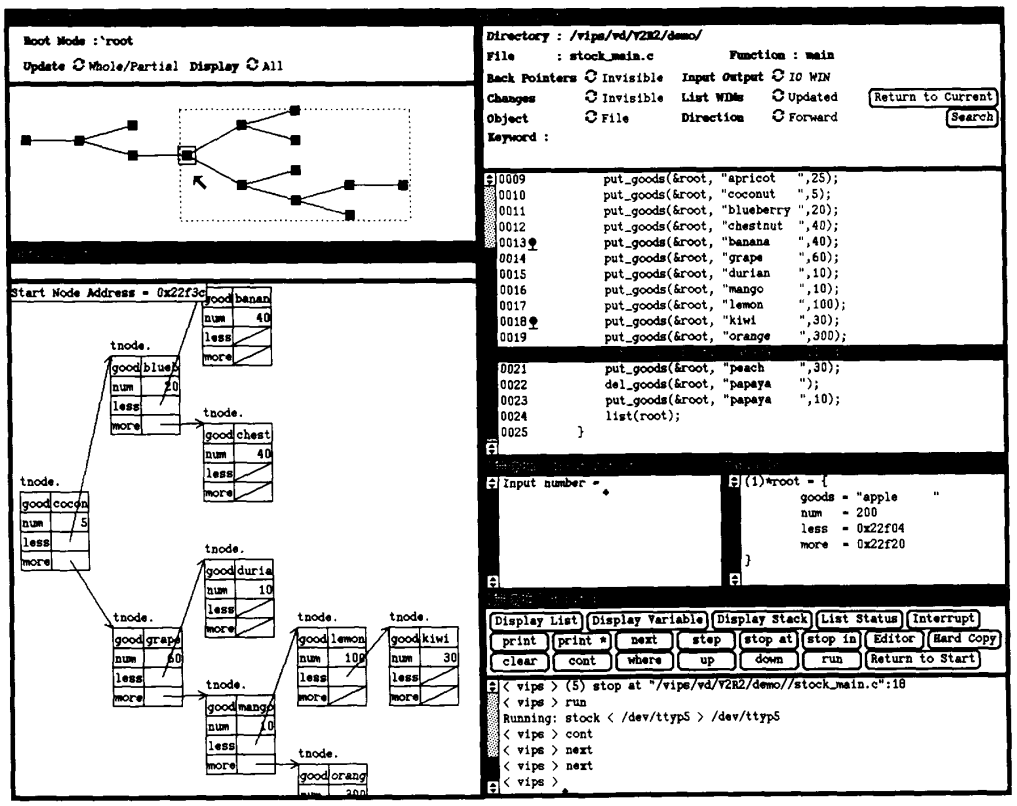
Displays the list structure as rectangles and arrows. Visualization facilities of list structures are described in the next subsection.

**(4) Input-output window:**

Displays input data to the program being debugged and its output data. A user can select either the Monitor or Input-output window for displaying input and output data.

**(5) Editor window:**

The source file of the program being debugged can be edited in this window. By clicking the **Edit** button with the mouse, an Editor window is opened and an editor specified in advance is invoked. The source text which is the same as the one in the Program-text window is displayed in the Editor window.




VIPS visualizes program execution from various aspects using multiple windows.

Fig. 1 VIPS screen example

**(6) Variable display window:**

Displays the values of specified variables every time the execution of a program is intermitted.

**(7) Stack display window:**

Displays the call stack of the program being debugged. By clicking the name of one of the functions in the Stack display window, the source program statement which invoked the function is shown highlighted in the Program-text window.

### 2.2 List Structure Visualization Facilities

#### (1) Requirements for List Structure Visualization Facilities

There are a certain kind of programs that allocate and manage inter-related data. These programs employ list structures for representing the data in the computer. For example, operating systems generate and manage various kinds of control tables as list structures; compilers deal with syntax tree tables; editors deal with tables that manage the lines displayed on the screen; hypertext systems deal with networks consisting of text, figures, etc.; and software repository systems deal with various kinds of information processed during software development.

Because list structures are dynamically allocated during program execution and can be complicated, their programs are often difficult to debug. Therefore, some techniques are required for facilitating the debugging process of programs that use list structures. For this purpose, visualization of list structures seems to be promising. However, it has to satisfy the following requirements.

( Requirement 1 ) List structure shape must be easy to recognize.

If a list structure has an erroneous pointer value, it results in an erroneous shape. Therefore, in order to find such errors as early as possible, it is very important that the list structure shape be easy to recognize.

( Requirement 2 ) Selective display of list structures must be possible.

If a list structure is very large and consists of several different types of nodes, a facility is required for selecting the part that a user wants to see from the large list structure and then displaying them. It is also necessary that the value of each node element in the list structure can easily be referred to.

( Requirement 3 ) Changes in list structures can be easily detected.

When an error occurs in a list structure, it is very important not to overlook it. Therefore, it is necessary to be able to quickly notice changes in either the list structure shape or values for the data elements of each node.

( Requirement 4 ) List structure drawing must be rapid.

Fast response time is very important for efficient debugging. Therefore, it is a requisite that a visual debugger can draw list structures rapidly.

The following two subsections describe VIPS list structure visualization and rapid drawing facilities that satisfy these four requirements.

#### (2) List Structure Visualization Facilities

**(a) Whole/partial list structure display facility**

When a user specifies a variable as the root node and instructs the VIPS system to display a list structure, two list windows are opened. They are a Whole list window and a Partial list window. They play the role of displaying the global and local views of a list structure, respectively ( Fig. 1 ).

- **Whole list window**

This displays a global view of the whole list structure that is governed by the root node specified by a user. The global view is a simplified, skeletal form consisting of just small boxes connected by arrows. Nodes are drawn in the depth-first order beginning from the root node. The location of each node is determined in the postorder by the layout algorithm described in section 3. The list structure figure in the Whole list window looks like a tree, because links pointing at previously drawn nodes are not shown ; they are only shown when specifically requested.

The root node can be specified in any of the following ways.

① Select the name of a variable in the Program-text window by clicking it with the mouse and then choose a **Display list** command from a button or menu.

② Locate the mouse cursor on a pointer field of a node displayed in the Partial list window, and then choose a **Display list** command from a menu.

③ Input a **Display list** command from the keyboard together with the address of a node and its data type.

- **Partial list window**

This window displays the selected part of the list structure in the associated Whole list window in detail. The selection is performed by clicking a node in the Whole list window. Then that node is set as the current **start node** for the Partial list window. The Partial list window displays the part of the list structure that is enclosed by a rectangle with the **start node** at the middle left, where the size of the rectangle is determined by the window size and the size of a node in the Partial list window. The Partial list window can display the values of the elements contained in each node.

**(b) Element value display facility**

In the Partial list window, the value of each element contained in a node is shown in a rectangle. A rectangle for each node can display values of at most five elements. Therefore, if a node is a large data structure, only the elements in its upper levels are displayed with the remaining elements shown in a reduced form ( Fig. 2(a) ).

There are two ways to see the values of reduced elements.

① Locating the mouse cursor on the reduced node, choose a **Display node** command from a menu. Then a Node display window is opened and it displays the values of all the elements of that node ( Requirement 2 ).

② Locating the mouse cursor on a reduced node, choose a **Select element** command from a menu and enter the names of the elements which a programmer wants to see. Then the values of the specified elements are displayed in the rectangle.

**(c) Updating list structure facility**

List structure figures in list windows are updated when the program execution is intermitted; this occurs when the program control comes to a breakpoint or every time a statement execution is finished in the stepwise execution mode, and so forth. Because VIPS(V1) updates figures each time a statement

is executed, its execution speed is very slow, and therefore it can be used only for algorithm programming education where execution speed does not greatly matter. For practical usage, however, programs must be executed as fast as their binary programs. Therefore, we have decided to update figures only when the execution stops.

**(d) Selective display facility**

In the case of a large and complex list structure, many nodes of several different types can be linked together. We have realized a facility that can selectively display only the part of the list structure linked by selected pointers. This facility enables a user to easily recognize the part of the data structure to be checked ( Fig. 2 ) ( Requirement 2 ).

**(e) Correspondence facility**

A user is allowed to open multiple list windows for displaying a list structure from various aspects. Thus a facility for relating corresponding nodes in two or more different list windows is required. Locate the mouse cursor on a node displayed in a certain list window, and then choose a **Corresponding node** command from a menu. This causes corresponding nodes in all the opened list windows to start blinking until the list windows are updated.

**(f) Highlight facility**

In a list window, elements whose values have been changed are highlighted. Even when the change in a list structure is small, this facility can help a user find it ( Requirement 3 ).

**(3) Rapid Drawing Facilities**

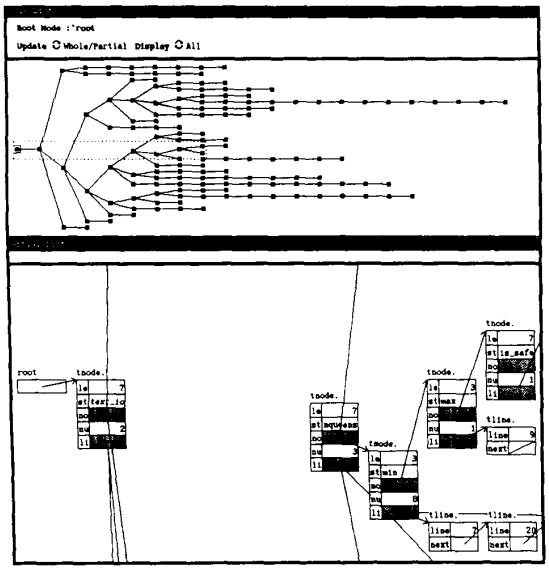
When a list structure is large, drawing all the nodes of the structure requires a lot of time. Therefore, it is necessary to be able to display only the nodes necessary for debugging. In addition to the selective display facility which also serves this purpose, we have developed the following facilities ( Requirement 4 ).

**(a) Setting number-of-layers facility**

This facility limits the number **n** of layers displayed in Whole list windows. Only nodes located in the first **n** layers of a list structure are displayed.

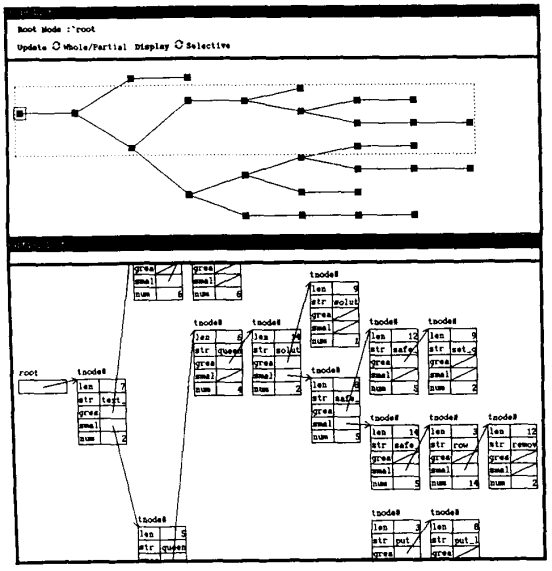
**(b) Partial updating facility**

When a list structure is changed, both the Whole and Partial list windows are updated. However, it is often the case that only the nodes in the Partial list window change but the shape of the list structure in the Whole list window remains the same. This situation can often happen in the stepwise execution. The partial updating facility utilizes this characteristics for attaining fast execution. When the partial update mode is specified, the Whole list window is left unchanged and only the Partial list window is updated. It is appropriate to apply this facility only for the case that changes in the list structure only occur within the Partial list window. Using this facility, the list structure can be updated very rapidly because only the nodes displayed in the Partial list window have to be updated. For example, when a list structure consists of 100 nodes, the drawing speed is about six times faster than when the Whole list window is updated as well.




In the case of a large and complex list structure, many nodes of several different types can be linked together.

Fig. 2 (a) List structure selective display




The list structure is selectively displayed by selecting only the nodes that are linked by pointers, Greater and Smaller. Unnecessary parts disappear and the displayed list structure becomes simple.

Fig. 2 (b) List structure selective display

## 3. Implementation

### (1) VIPS System Configuration

VIPS has been developed on SUN 3 workstations. VIPS makes use of dbx, a debugger provided by Sun microsystems, for execution of programs to be debugged. VIPS sends a command to dbx via a pipe, receives a response and analyzes it, and then displays the execution in multiple windows. Dbx's debugging commands are still available from the VIPS system. The VIPS configuration is illustrated in Fig. 3.

### (2) Interface with dbx

Conversation between VIPS and dbx for drawing a list structure is the most important interface with dbx. For each pair of list windows, the following procedure is used by VIPS to create a **List structure table** which contains the types and values of all the nodes in a list structure ( Fig. 4 ).

① Acquiring the types and values of the root node

VIPS acquires the type information of the root node from the symbol table of the program being debugged. Then it sends a command ( *print root* ) to dbx and acquires the values of the elements in the root node.

② Acquiring the types and values of other nodes

VIPS investigates the type information of the root node to find pointers included in the root node. Then it acquires their types ( for example, *struct tnode ** ) and values ( *0x22ef4* ). Next, it sends a command ( *print *(struct tnode *)0x22ef4* ) to dbx to acquire the values of the nodes pointed at by the pointers. By repeating this procedure, it acquires the types and values of all the nodes necessary to draw a list structure.

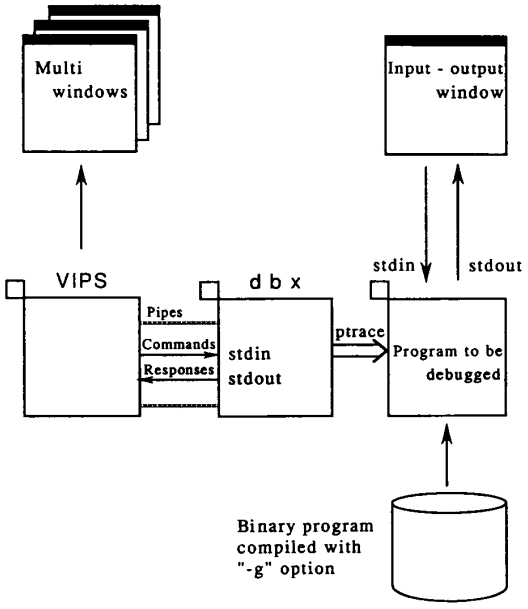
### (3) Layout Algorithm

The layout of a list structure is performed by applying the following procedure call to the root node of a given list structure,

layout ( root, 0, root_y ),

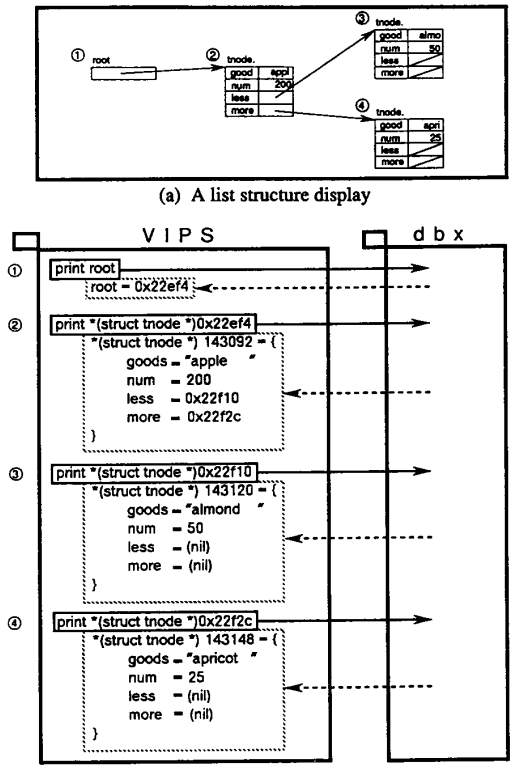
where **root** gives the root node address, **0** indicates the layer level of the root node, and **root_y** will return the value of the y-direction coordinate of the root node in the list structure figure.

The procedure **layout** is shown in the Appendix. It recursively applies itself to each of the child nodes of the **node** in the first parameter. Nodes in the list structure are allocated in the postorder. For facilitating recognition of the list structure,




VIPS makes use of a debugger dbx on SUN 3 workstations. VIPS sends a command to dbx via a pipe, receives a response, analyzes it, and then displays the execution in multiple windows. Dbx's original debugging commands are also available.

Fig. 3 VIPS Configuration




(a) A list structure display

(b) Conversation between VIPS and dbx for drawing the list structure in (a)

Fig. 4 Example Interface with dbx for drawing a list structure

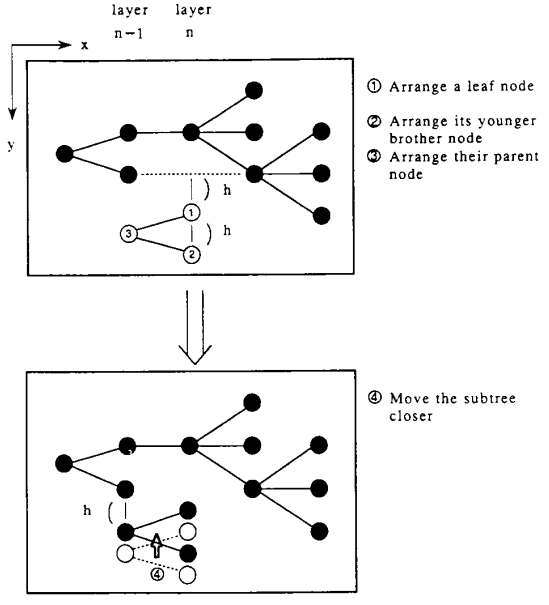
each branch node is located at the middle of its child nodes in the width-direction of the list structure ( i.e., y-direction ). This satisfies Requirement 1. Every time a branch node is allocated, the subtree starting from the node is checked to see whether it can be moved upward to optimize the distance between itself and the previously allocated subtrees. This algorithm enables the list structure to be laid out in a minimum amount of space on the condition that parent nodes are centered at the middle of their child nodes ( Fig. 5 ).

## 4. Evaluation

### (1) Efficiency for Drawing List Structures

Because a binary program is executed, VIPS can execute programs at a far more rapid speed than VIPS(V1) which uses an interpreter. A preliminary evaluation is performed on the efficiency for drawing a sample list structure which consists of nodes with two pointers and one string element. The time **t** needed to update the figure of the list structure is given in the formula below ( Fig. 6 ).

$$\left.\begin{aligned} \mathbf{t} &= \mathbf{t1} + \mathbf{t2}, \\ \mathbf{t1} &= 30\ (\text{ms}) \times \mathbf{n}, \\ \mathbf{t2} &= 10\ (\text{ms}) \times \mathbf{n} + 40\ (\text{ms}) \times \mathbf{m}, \end{aligned}\right\} \quad (A)$$




VIPS allocates nodes of a list structure in the postorder. Every time a parent-node is allocated, the subtree starting from the node is examined as to whether or not it can be moved upward. This algorithm can draw the list structure in a minimum amount of space.

Fig. 5 Layout algorithm for list structures

where

**t1** is the time to create a **List structure table**,
**t2** is the time to draw the list structure,
**n** is the number of nodes in the list structure and
**m** is the number of nodes displayed in the Partial list window.

Even for a large list structure, the number of nodes which a user wants to see at a time may be less than one hundred. Putting one hundred into **n** and ten into **m** in the formula (A), **t** becomes about four seconds. This shows that it is necessary to limit nodes to be displayed in order to accomplish practical speed for displaying list structures as figures. One way to do this is to use the partial update facility. The time **t** needed to update only the Partial list window is given in the formula below.

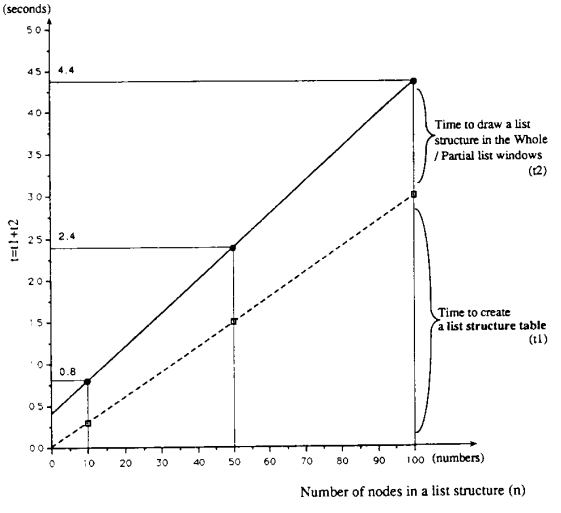
$$\left.\begin{aligned} \mathbf{t} &= \mathbf{t1} + \mathbf{t2} \\ \mathbf{t1} &= 30\ (\text{ms}) \times \mathbf{m} \\ \mathbf{t2} &= 40\ (\text{ms}) \times \mathbf{m} \end{aligned}\right\} \quad (B)$$

When ten nodes are displayed in the Partial list window, the time **t** for updating a list structure displayed in the window becomes only 0.7 second.

About fifty percent of the time **t1** to create a **List structure table** is spent by dbx's command processing and the communicating between VIPS and dbx.

### (2) Effectiveness of List Structure Visualization Facilities

In order to see the effectiveness of list structure visualization facilities, the authors performed a preliminary experiment.




The graph shows the time required to display a list structure with **n** nodes of which **m** ( = 10 ) nodes are displayed in the Partial list window.

Fig. 6 Time required to display a list structure

Several programs were debugged using VIPS and dbxtool [1] which is a character-based symbolic debugger with source text display facility.

**(a) Debugging experiment**

Two programs are prepared which deal with list structures. One of the programs analyzes numerical expressions by creating a syntax tree. Another program handles a binary tree. Both programs are written in C language and are about 200 lines of code in size. Several bugs are buried in the programs in advance. Two programmers debug them using each of the two debugging tools. The time required to detect the bugs and the debugging procedure are recorded. Here the time to detect a bug means the time a programmer spent finding a bug, and the debugging procedure means what a programmer did in debugging, such as what kind of errors he noticed during the execution, what he wanted to know next and what commands he used.

In the experiment we paid attention to the following points.

① If a programmer records his debugging procedure while he debugs a program, the exact time required to detect a bug cannot be measured. Therefore, we performed the experiment by repeating the following two steps; in the first step a programmer debugs a program until he finds a bug without keeping any records, and in the second step he repeats the same debugging process that he did in the first step while taking records of the procedure.

② If a programmer were to find more than one bug at the same time, it would be difficult to estimate the time required to detect each of the bugs. Moreover, if a program should contain more than one bug and a programmer had to find them all, he would have to fix a bug and then proceed. But this might result in degradation of the program and the experiment results might be meaningless. Therefore, we prepared six versions of the programs containing just one bug for each of the two programs.

**(b) Experimental results**

The average time to detect a bug is 6.4 and 9.5 minutes for VIPS and dbxtool, respectively. The average number of commands is 16 and 22 for VIPS and dbxtool, respectively. This result means that VIPS can reduce debugging time by about 30 percent and the number of debugging commands by about 25 percent.

The reduction in the number of commands comes from the difference in the number of commands required to display data values. While VIPS required a programmer to use display commands 2.3 times, dbxtool required 8.0 times in the average for finding a bug. This is because, when a programmer wants to see whether or not a list structure has been correctly created, he has only to perform a **Display list** command once with VIPS, while he has to use print or display commands as many times as the number of data elements contained in the list structure with dbxtool.

## 5. Conclusion

VIPS is a visual debugging system which makes it easy to debug programs by visually displaying the program execution from various aspects using multiple windows. It has various facilities that can help a user understand complicated list structures with ease. A preliminary evaluation shows that VIPS can greatly improve debugging efficiency.


## Acknowledgments

We are grateful to the late Dr. Shuetsu Hanata of NTT Software Laboratories for his guidance and encouragement. We also wish to thank Kenji Takahashi and Shouichi Kondou for their useful advice and cooperation.


## Appendix

```
procedure layout( node: pointer; level: integer; y: in out
integer ) is
   h: constant integer:= { the height of a node };
begin
   if { node has no children } then          -- a leaf node
      y:= max( y-coordinates of the previously allocated
      nodes in layer 0 through level ) + h;
      node.x:= level;
      node.y:= y;
   else                                      -- a branch node
      for { each child node } loop
         layout( { a child node }, level + 1, child_y );
      end loop;
      y:= ( min( child_y's ) + max( child_y's ) ) / 2;
      node.x:= level;
      node.y:= y;
      -- If possible, move the subtree upward.
      margin:= min(
         min( the distance between node and each node in
         layer 0 through level ),
         min( the distance between each of the subordinate
         nodes of the subtree and its nearest node contained
         in the previously allocated subtrees in the same layer
         )) - h;
      if ( margin > 0 ) then
         y:= y - margin;
         { move the subtree upward by subtracting margin
         from y-coordinates of the subtree nodes };
      end if;
   end if;
end layout;
```

## References


[1] Evan Adams and Steven S. Muchnick, "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations," Software-Practice and Experience, July 1986, pp.653-669.

[2] J. D. Bovey, "A Debugger For A Graphical Workstation," Software-Practice and Experience, Vol. 17(9), September 1987, pp. 647-662.

[3] James H. Griffin, Harvey J. Wasserman and Lauren P. Mcgavran, "A Debugger for Parallel Processes," Software-Practice and Experience, Vol.18(12), December 1988, pp. 1179-1190.

[4] Steven P. Reiss, "PECAN: Program Development Systems That Support Multiple Views," Proc. 7th ICSE, 1984, pp. 324-333.

[5] Marc H. Brown and Robert Sedgewick, "Techniques for Algorithm Animation," IEEE Software, January 1985, pp. 28-39.

[6] Thomas G. Moher, "PROVIDE: A Process Visualization and Debugging Environment," IEEE Transaction on Software Engineering, Vol. 14, No. 6, June 1988, pp. 849-857.

[7] Brad A. Myers, Ravinder Chandhok and Atul Sareen, "Automatic Data Visualization for Novice Pascal Programmers," 1988 IEEE Workshop on Visual Languages, October 1988, pp. 192 - 198.

[8] D. P. Pazel, "DS-Viewer - An interactive graphical data structure presentation facility," IBM SYSTEMS JOURNAL, Vol. 28, No. 2, 1989, pp. 307-323.

[9] S. Isoda, T. Shimomura, and Y. Ono, "VIPS: A Visual Debugger," IEEE Software, Vol. 4, No. 3, May 1987, pp. 8 - 19.