Design of a Debugger for Large-Grain Data Flow Programs

David C. DiNucci

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-005

December, 1988

Design of a Debugger for Large-Grain Data Flow Programs

David C. DiNucci

Department of Computer Science and Engineering
Oregon Graduate Center
19600 NW Von Neumann Drive
Beaverton, OR 97006 USA
(503) 645-1121

Abstract — **Most approaches to parallel debugging have assumed the worst: that the program to be debugged must be viewed as a black box, with no internal structure useful to the debugger. This document assumes the opposite—that the program was written using Large-Grain Data Flow 2 (LGDF2). Using this as a basis, the design of a user interface for a graphical debugger is proposed, to view and control execution of a parallel program from a high, inter-process level before descending into a standard debugger to view the intra-process actions. Breakpoints are presented in a natural way which doubles as a single-step mechanism. LGDF2 provides the framework to reliably repeat executions, to restrict non-breakpointed processes from altering important state, and to facilitate pre-execution analysis to determine the minimum amount of data that needs to be logged during tracing.**

Index Terms — **Parallel Debugging, Instant Replay, Non-determinism, Large-Grain Data Flow, Graphical Debugging, Program Visualization, High-level Debugging, Parallel Monitoring.**

1. INTRODUCTION

A program written using the Large-Grain Dataflow model (LGDF2)[1] is highly structured, both syntactically and semantically. LGDF2 can be viewed as a declarative very-high-level language, where the atoms are themselves processes written in a sequential high-level language. This project is to test two hypotheses relating to debugging an LGDF2 program:

(1) Debugging a parallel program is often confounded by the nondeterminism present. Since the debugging environment is often quite different than the actual environment in which the program usually executes, especially considering the relative execution times of the communicating subtasks, bugs that are present during normal execution may dissappear during debugging. Many have attempted to solve this problem by capturing the important events of an actual execution, using that event stream to replay the execution in the debugger. But just the time required to capture the events during the actual execution can cause problems. We believe that the structure of an LGDF2 program can be analyzed before execution to cut the number of events to be captured to an absolute minimum, thereby greatly decreasing the effect of this action on execution.

(2) Much of the debugging of parallel programs consists of determining what went wrong at a very high level; what process didn't start when it was supposed to, which communication didn't take place, why the effect of changing this data have the desired result over at that process. This type of debugging does not require the user to trace the internal process states, but rather the interaction between processes. Since, in an LGDF2 application, all such interaction takes place within the network and not within the processes, this type of debugging can occur without delving into the actual processes. A low-level debugger can still be made available to debug the individual processes for logic-errors at that level.

The LGDF2 debugger runs on a graphics workstation and controls and monitors the execution of a program which is running simultaneously on a parallel computer. At the user's request, a standard source-level debugger (on the parallel computer) can be invoked. The behavior of the LGDF2 debugger is intended to be independent of the workstation (running the debugger), the parallel computer (running the program and low-level debugger), and the debugger used for low-level debugging.

In the remainder of this document, Section 2 will describe how we exploit the LGDF2 model, Section 3 will describe the windowing features of the display, Section 4 will describe the annotations given to objects within those windows, and Section 5 will describe the actual dynamic behavior and semantics of the debugger.

2. Overview

This debugger depends on many of the features offered by the LGDF2 programming model. Though the description of this model is beyond the scope of this document, we will repeat the aspects which we feel are important in this context:

(1) An LGDF2 program consists of processes. For the purpose of this paper, these will be considered to be sequential program segments, written in a standard sequential language (e.g. Fortran or C) for which standard symbolic debuggers are readily available. These processes fire (begin execution) spontaneously, often repeatedly during a single run of the LGDF2 program, according to the following rules.

(2) LGDF2 processes communicate only through selectively-shared data spaces called datapaths. The permissions (read, write) that each process has to each datapath is declared as part of the program, usually by showing the processes as being connected with undirected, directed, or bidirected arcs to the datapath in a graphical form of the program called a network or data flow graph. A process accesses the data on the datapath as though it was a call-by-address parameter. Processes cannot maintain internal state between firings, but such state can be explicitly saved on a datapath to be re-accessed in a future firing.

(3) In addition to data, each datapath possesses a control state which defines the subset of processes which can access it at that point in time. When depicted graphically, the members of a given subset are represented by having them all connected to the same side of the datapath. A process can only fire when all of the datapaths to which it has permissions have control states which allow it access. Upon firing, a process will be guaranteed logically exclusive access to those datapaths. At any time, a process may relinquish that access, and at the same time alter the control state of the datapath to give other processes access. Access may not be regained unless the process fires again.

These rules allow us to regard each process externally as a function, deterministically mapping the values of its readable datapaths to new values for its writable datapaths and new control states for all of its datapaths. Internally, the sequential code comprising each process is guaranteed to act as though it was running stand-alone on a sequential computer. This debugger exploits this clean division by using a graphical interface to manipulate the high-level interactions, treating each process as a function, relegating the lower source-level debugging to any existing debugger already designed to manipulate sequential programs.

As an aid to both programming and debugging, this document will assume the ability to represent data flow graphs hierarchically, even though this has not been formally discussed in the context of previous papers on LGDF2. The principal is simple: a node on a dataflow graph (shown as a circle) can represent either a process (as discussed above) or another data flow graph in its own right. This data flow graph (or subnet) will have certain of its datapaths annotated to show that they are actually "off-page connectors" or formal parameters, and represent a datapath (actual parameter) within its parent network.

In addition to facilitating a natural division of labor in the debugger, the LGDF2 model also addresses the problems related to debugging non-deterministic programs. Since the processes are all deterministic, and since the relative firing order of processes accessing a single datapath is dictated partially by the control state of the datapath

which is in turn deterministically determined, certain portions of the LGDF2 program can be ensured to be deterministic by inspection [2]. To capture the behavior of the non-deterministic portions, it is only necessary to record the order that certain processes connected to the same side of a given datapath fired. This can be done with a minimum of overhead during a production execution, and the resulting record can be fed into the debugger to ensure that those processes fire in the same order during debugging[3]. When our debugger is looking at such a record and ensuring the same partial order of firings as in the production run, we will say that it is in "replay" mode.

For a more extensive overview of the leverage that LGDF2 can provide to a debugger, see [4].

3. Windows and their Static Contents

This section will describe the windows which a user may conjure up. The only dynamic actions described in this section will be the opening, closing, and possibly resizing of windows as well as scrolling through their contents. All of these actions are performed only as a direct result of action from the user. The user-interface should be as close to the standard u-i for the machine that the debugger is executing on as possible. Preferably, all actions should be able to be performed through the mouse alone.

The display originally consists of the main node window (for the highest level network) and one execution window. Four menu sections will also be accessible, but may be implemented as pop-up or pull-down rather than static. All elements of the original display will always either be somewhere on the screen or easily summoned.

The user may open and close other node windows through the use of the mouse and menu, as described below. [Optional - The user may open and close datapath windows through the use of the mouse and menu, as described below.]

Figure 1 is an example of a screen after opening a network node (fiddle) and a process node (fiddle.x) window, with the execution and main node windows at the left, and the menus along the bottom of the screen. This example will be referenced as its different components are discussed.

3.1. Node windows

Each node window will have a standard title bar, which will contain room for a string of up to 5 8-character node names separated by periods [optional: and a one-line node description]. The main node window, with an empty title bar, will be a network node window and will thus contain a graphics image of an LGDF2 network constructed by a separate graphics editor. This network will consist of nodes, datapaths, and arcs, as per LGDF2 standards (see top of Figure 2 for details):

- Each node will be represented as a circle (or many-sided polygon) labeled with an 8-character name and, optionally, a small integer making the name unique within the window. [Optional: Process nodes are distinguished from network nodes.]
- Each datapath will be represented by a vertical rectangle labeled with an 8-character name.

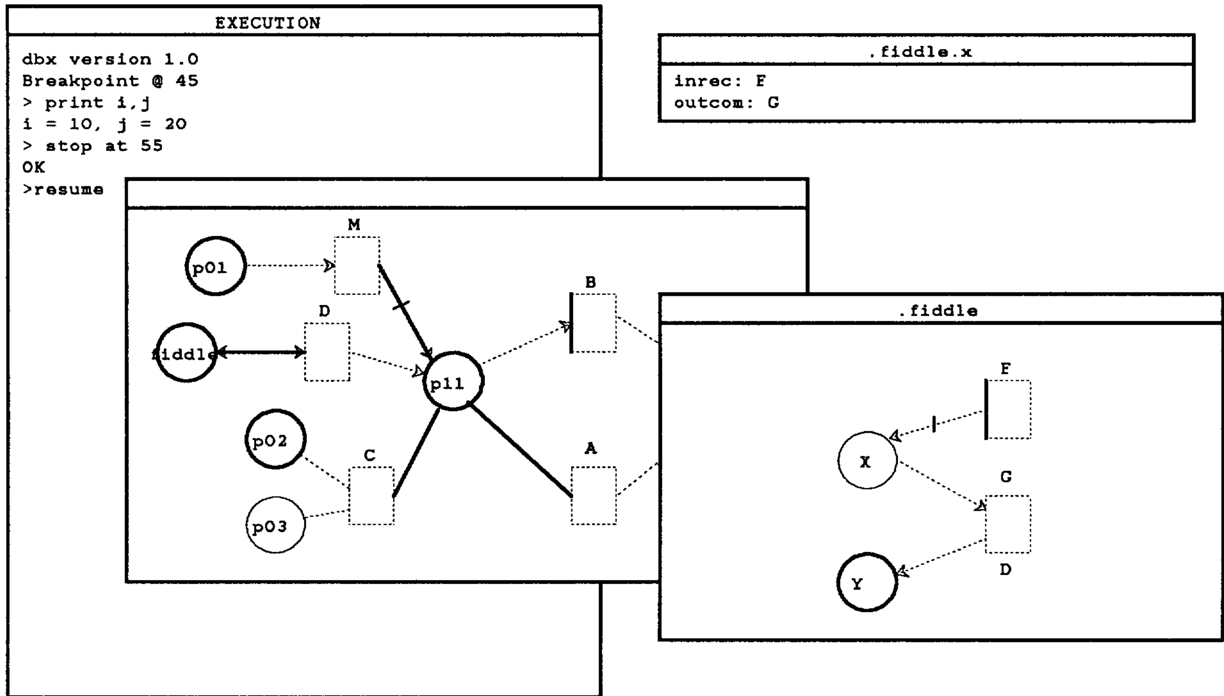


Figure 1. Sample Screen.

- Each arc will be represented by a line connecting either the left or right side of a datapath to a node. The arc may have an arrowhead at both, either, or neither end. The arc may be either grantable or ungrantable; in the latter case, it will be annotated, preferably with a dot, cross, or slash, across the arc somewhere along its length.

Each subsequent node window must be associated with a node in an existing node window, and its title bar will contain the name of the node shown therein prefixed with the title of its parent window. The new window will be called a subwindow of its parent.

- If the node is a network node, the (network) node subwindow will will contain an LGDF2 network of the form described above except that it will have certain of its datapaths marked as "external", one for each datapath adjoining the node in its parent window. These external datapaths will be distinguished within the subwindow by being labeled with the name from the parent window as well as their own name. This additional (external) name should be easily differentiated from their own (internal) name, either by its location or type font. Window

`.fiddle` in figure 1 is a (network) node subwindow of the main node window.

- If the node is a process node, the (process) node subwindow will contain a list of internal-external datapath associations. This can be represented textually, or graphically as if it was another network node subwindow containing only one node. Window `.fiddle.x` in figure 1 is a (process) node subwindow of (network) node window `.fiddle`. [Optional: In addition to the internal-external datapath associations, the (process) node subwindow contains a second pane with a listing of the process source code. Some means is available to scroll through the process code - preferably with the mouse only, like a scroll bar.]

All nodes with the same name (regardless of occurrence number) in a common network (i.e. window) will have node subwindows which are identical, with the possible exception of having different external names.

3.2. Execution Window

The execution window may be on a separate screen, in which case it will probably have a devoted keyboard. It is used to facilitate communication between the low-level (source-level) debugger and the user. It may also be used for terminal I/O to/from the user's parallel program being debugged. The window will likely not require graphics capability. The LGDF2 debugger does not perform any I/O directly to/from this window. Therefore, this window will only be described in vague terms in this document. In figure 1, the execution window, `EXECUTION`, is on the same screen as the other windows.

3.3. Menu

The menu contains four sections - Node/Path Action, Global Action, Display Speed, Global Mode, and Window Action. The first four of these are shown in the long rectangle at the bottom of the screen in figure 1.

- The Node/Path Action section is used to select the accion to be invoked on an object in a node window (i.e. a node or datapath) when picked with the mouse. The menu can be implemented either as a static menu or a pop-up menu. For a static type menu, the user would first choose a menu item, and any subsequently picked object would have the chosen action applied to it; a menu item would remain chosen until another item was chosen, and the mouse pointer should change (if practical) to reflect the currently selected item. The Node/Path Action section in Figure 1 is implemented in this way. For pop-up, the menu would appear at the mouse location whenever the user depressed the mouse button within an object, then whichever menu item the mouse was over when the button was released would be picked and that action would be applied to the object. There are currently five menu items - "Freeze/Thaw/Unstick", "Print/Debug", "Tag/Zap/Untag", "Open Window", and "Timed Freeze". The "Freeze/Thaw/Unstick" menu item will be used quite often, so systems which have multiple mouse buttons may dedicate one of them to performing only this action.

- The Global Action section allows the user to apply the "Freeze/Thaw/Unstick" to all datapaths or processes on the screen which currently have a certain temperature (explained later), or the "Tag/Zap/Untag" action to all datapaths or processes on the screen which currently have a certain trace mode (explained later). It consists of six items, three which resemble process circles and three which represent datapath rectangles. Either the color, shading, or annotations on these items will change as they are selected. Though this may be implemented as a pull-down or pop-up menu, it is likely that the implementation of shapes and changing colors or shadings will require that it be implemented as a static menu. The exact functioning of this menu will be described later.
- The Display Speed section alters the speed at which events trace events are displayed on the screen. It consists of a "Stop" item, several speed items (marked with approximate number of events per minute), and a "Hyper" item. Only one of the items on this menu can be selected at any one time, and the selected item remains selected until another item is chosen. The speeds available may differ depending on the debugger implementation.
- The Global Mode section alters the function of the entire debugger. When an item from the menu is chosen, it must remain selected until it is chosen again. It may be implemented as a static menu, a pull-down menu, or a pop-up menu (when the mouse button is depressed outside of an object), and may share a menu with the Global Action menu. Any, all, or none of the menu items may be chosen at any one time. The menu currently contains only one menu item - "Replay". Once this item is deselected, it can never be selected again.
- The Window Action section consists of those actions already supported by most windowing systems - "Close window", "Move window", "Resize window". These are usually supported by gadgets or pull-down menus in the window border. This should be implemented in the way supported by the window system being used.

3.4. [Optional: Datapath windows]

Each datapath window must be associated with a datapath in an existing node window, and its title bar must contain the name of that datapath. The window contains a listing of the datapath structure (i.e. source code of type declarations for variables on the datapath). Some means to scroll through the contents (prefereably with the mouse only, like a scroll bar) is also required.

4. Debugging Annotations

This section will describe annotations which are added to a node window while an LGDF2 program is executing. Three line styles are used - dotted (or dashed), thin (or normal) solid, and thick (or highlighted) solid. If thick lines are not available or do not stand out enough, they can be thickened by using doubled lines. Either shading (none, light, and heavy) or color (green, yellow, and red) will be used to fill objects. The choice of whether to use shading or color for an implementation will be a function of speed and capabilities of the display. Other annotations are formed of characters

added to the display.

These debugging annotations fall into four general categories - temperature, execution/data state, debugging/printing state, and dataflow tracing. A possible representation for each annotation described in this section is shown in figure 2.

4.1. Temperature

Each object (node and datapath) has a temperature which is reflected by its color or shading. A Thawed object is green or unshaded, a Frozen object is yellow or lightly shaded, and a Stuck object is red or deeply shaded. An arc has no temperature, and is always shown in green except as noted in the next section.

Each process also has a "firing count", which is a one-to-five digit number. This can be used to alter temperature through the "Timed Freeze" facility described later.

4.2. Execution/Data State

Each process has an execution state and each datapath has a data state, reflected by the line style of its outline. Arcs have usage states which are also shown this way.

- A process's execution state can either be Suspended (idle), in which case the circular outline is dotted or dashed, Fired (but not executing) in which case the circular outline is a thin solid line, or Executing in which case the circular outline is a thick or highlighted solid line.
- A datapath's data state can either be Neutral (none), in which case the rectangular outline is a dotted or dashed line, Left in which case the left vertical line of the rectangle is a thick or highlighted solid line while the rest of the rectangle is dotted or dashed, or Right in which case the right vertical line is thick or highlighted.
- An arc's usage state can either be Finished, in which case the line is dotted or dashed, or Unfinished, in which case the line is a thick or highlighted solid line. As described later, it is sometimes desireable to temporarily show when a process has attempted but failed to change the usage state of an arc. If color is available on the display, this is best shown by changing the color of the arc to red. However, if color is not available, the line style of the arc should be changed to a thin solid line. The actual usage state of the arc in this case will hopefully be obvious to the user from the surrounding cues.

4.3. Debugging/Printing State

Processes may be in "Waiting for debugger" state in which case they are annotated with a "D", "Has been debugged" state in which case they are annotated with a "d", or "Never been debugged" state in which case they are not annotated with either.

Datapaths may be in "Being printed" state in which case they are annotated with a "P", or "Not being printed" state in which case they are not.

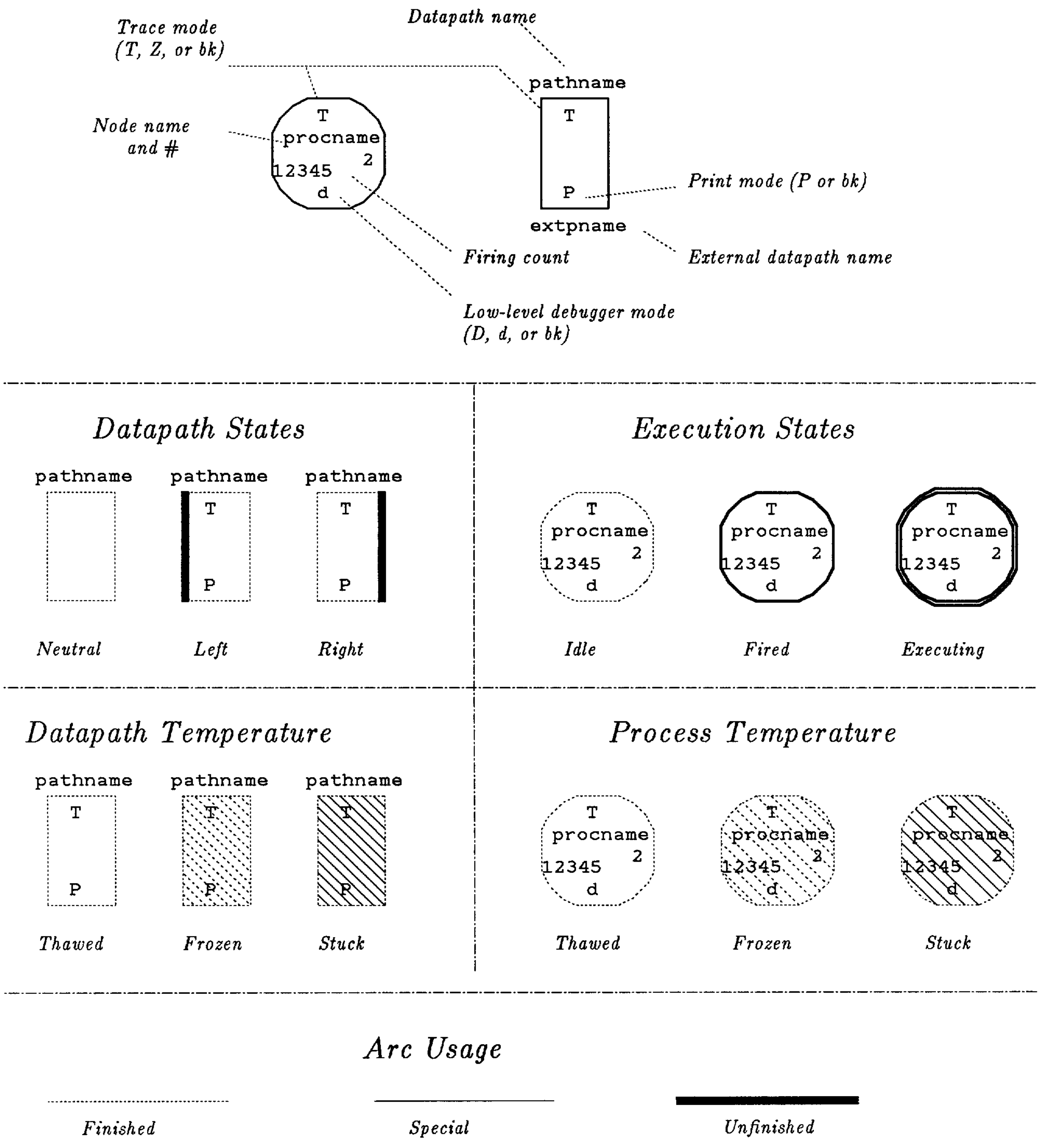


Figure 2. Possible Representations for Annotations

4.4. Dataflow Tracing

Data in an object (process or datapath) may be tagged, in which case the object is annotated with a T, or zapped, in which case it is annotated with a Z. Only one of these can be present at any one time. Any kind of extra color or highlighting for these

annotations is desireable to make the marked objects stand out as much as possible, but it must not conflict with any other markings already mentioned.

5. Debugging and Monitoring

This section will describe the actual functioning of the LGDF2 debugger from the user's standpoint. Justification of certain design decisions will be included.

Debugging will be described in six sections - getting started, high-level monitoring, stopping and starting processes, low-level debugging, dataflow tracing, and other menu actions.

5.1. Getting Started

When the debugger starts, it queries the user for the name of the program to be debugged and for the name of the trace information file produced during the normal execution of that program. The debugger then presents the initial windows for the program, as described under section 2, above. The "Stop" item on the "Display Speed" menu section is selected, so the LGDF2 network is not executing in any way. All processes are idle (i.e. outlines are not highlighted), all datapaths are Left (i.e. their left side is thick or highlighted) and all arcs are Finished (i.e. their lines are not dotted or dashed). If a filename was supplied at the trace information file prompt, the "Replay" menu item is selected on the "Global Mode" menu. By selecting the a Display Speed other than "Stop", the network will begin execution.

5.2. High-Level Monitoring

During the network execution, processes in the network may fire, execute, grant and reserve datapaths, and suspend. Because these actions will occur quite frequently and account for the vast majority of the animation process, their animation must be fast. For this reason, line styles (rather than color or shading which are rather slow on some displays) is used for all such animation, as follows:

5.2.1. Fire

According to the semantics of LGDF2, a process which can fire will have each of its arcs connected to the highlighted side of a datapath, as shown in figure 3.A. The act of firing will appear in 3 stages, which will hopefully occur in quick enough succession to give some feeling of animation. In the first stage, the arcs connecting the process to its datapaths will become unfinished (thick solid lines or highlighted), as shown in figure 3.B. In the second stage, the process execution state will become Fired (the outline will become thin-solid), as is shown in figure 3.C. Finally, the datapath states will become neutral (making the sides dotted or dashed) as shown in figure 3.D. The "firing count" on the process will also be incremented at this time.

If the "Replay" mode is selected, processes which appear to be able to fire (i.e. all of their datapaths are in the correct state and they are not executing) may not. This is because, in the execution that is being replayed, the firing of that process was preceded by the firing of another process that could have some effect on it, and that other

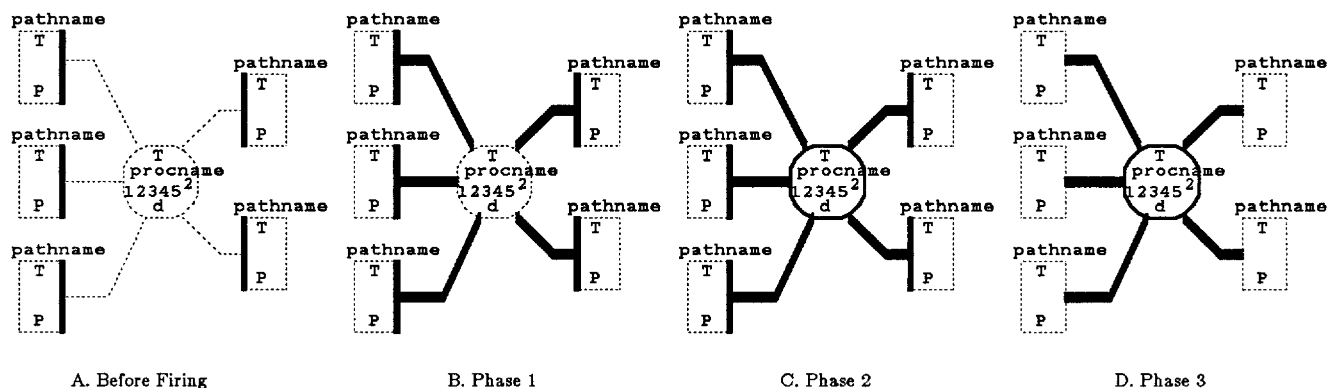


Figure 3. Phases of Firing Animation

process has not yet executed in the replay. The user may cancel replay mode at any time, but it may not be restored after it is canceled since executions may already have occurred in an order different from that of the original execution.

5.2.2. Execute

When a process state goes from "Fired but not executing" to "Executing", the outline of the process will be redrawn with a thick solid line. The debugger may, in some cases depending on the scheduler, merge the "Fire" and "Execute" phases into one to avoid drawing the process outline twice when not necessary.

5.2.3. Grant or Reserve a Datapath

While a process is executing, it may reserve or grant any datapath which is connected to the process via an unfinished (highlighted) arc. From the semantic properties of LGDF2, the datapath will always be Neutral in this case (i.e. be represented with dotted sides). These actions consist of altering the datapath state (shown by redrawing one side of the datapath with a thick or highlighted solid line), then finishing the arc (by redrawing it with a dotted line).

5.2.4. Suspend

A process which is executing and which has no unfinished datapaths may suspend. This is shown by returning it to idle state (i.e. by redrawing the process circle with a dotted line).

5.3. Starting and Stopping Processes

The actions discussed in this section are either the result of a mouse click or they require a mouse click before further animation can proceed, so the resulting animation does not need to be lightening fast. However, since some of the actions require user attention, they must be immediately obvious to the user. For these reasons, the coloring or shading of regions is used for most all actions discussed in this section.

5.3.1. Freezing Processes

One method of starting and stopping processes is by altering their temperature (represented by their color or shading). A thawed (green or unshaded) process will execute freely, while a frozen (yellow or lightly shaded) process will not grant or reserve any datapaths or begin execution after firing. The temperature of any process can be toggled from thawed to frozen by applying the "Freeze/Thaw/Unstick" menu action to the process.

If a process attempts to perform an action while it is frozen, it will become stuck (red or deeply shaded). If this was a result of attempting to grant or reserve a datapath, that datapath will turn also turn red if the display is in color or change from a thick/highlighted solid line to a thin solid line if the display does not have color. Applying the "Freeze/Thaw/Unstick" menu action to the process will allow it to execute the action which it became stuck on and return to a frozen state and the arc back to green (if it was red). By applying the same menu action again, while the process is frozen, it will of course become thawed as described before.

Occasionally, a process will perform dataflow actions in very quick succession, which would make it difficult to return it to a thawed state from a stuck state (because it wouldn't stay frozen long enough to apply the menu action the second time). For this reason, the debugger will guarantee that the temperature will not change too quickly to be caught by two mouse clicks within a 1/2 second interval.

A process can be set to freeze after firing a specific number of times by applying the "Timed Freeze" action to the process. This displays the current "freeze time" and allows the user to alter it. When the "firing count" of a process equals its "freeze time", the temperature will be set to frozen as it is firing. This will cause the process to become stuck before executing its first statement. If the debugger is in Replay mode, entering a zero or negative freeze time will cause the freeze time to be set to the last "firing count" recorded for the process during its regular execution minus the absolute value of the freeze-time entered.

Freezing a process can be used to single-step the dataflow actions within it for careful scrutiny, to see the effect of stopping it on the surrounding processes, or to temporarily halt it to engage a lower-level debugger. Timed freezing allows the user to quickly advance to interesting portions of the execution. It should be noted that freezing a process does NOT stop it from firing.

5.3.2. Freezing Datapaths

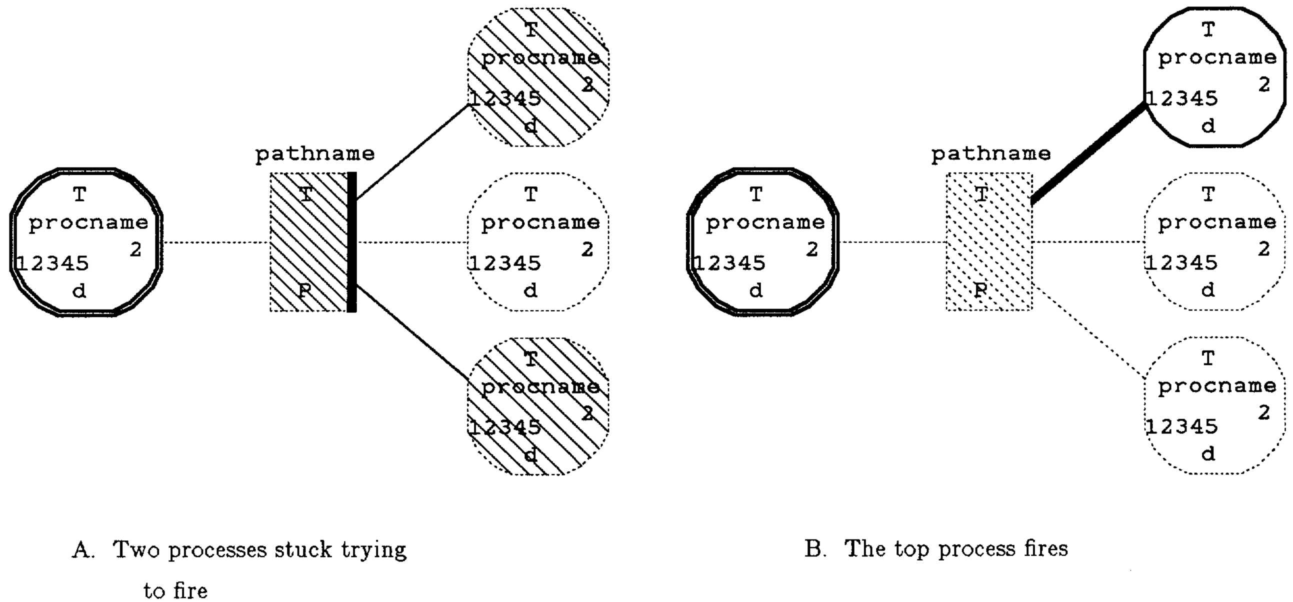
The other method of stopping and starting processes is by altering the temperature of their datapaths, which is shown through color or shading in the same way as for processes. Any process connected to a thawed datapath can fire freely, according to the laws of LGDF2, while a process connected to a frozen datapath is not allowed to fire. When such a process attempts to fire, both the datapath and the process become stuck and the arc connecting the datapath to the process becomes red (if color is available) or thin-solid (if not). Even after the datapath has become stuck, it will still cause other processes connected to it which attempt to fire to become stuck and

for their connecting arc to become red or thin-solid.

Applying the "Freeze/Thaw/Unstick" menu action to the datapath will allow one of the frozen processes (chosen by the scheduler) to fire. This will return all processes stuck on the datapath to their previous temperatures[1], since they will no longer be stuck, only idle. The temperature of the datapath will also therefore return to frozen. Figure 4 illustrates what happens when a stuck datapath is unstuck while two (previously thawed) processes are stuck waiting for it.

Applying the "Freeze/Thaw/Unstick" menu action to one of the stuck processes will cause it to fire and change its temperature to frozen. All other processes and the datapath will act as described above. This allows the user to dictate which process is to fire next.

Freezing a datapath is useful for holding it in a stable state (i.e. between process accesses) to view its contents or for capturing interprocess interactions. A global datapath freeze can be used as a network-wide single-step, where "step" here is defined as a process firing. If the debugger is not in "Replay" mode, this also allows the user to pick which of the processes competing for the datapath should go first.



A. Two processes stuck trying to fire

B. The top process fires

Figure 4. Unsticking a datapath

[1]These constitutes the only hidden modes in the debugger - the temperature that the process will return to and the next process to execute. We may still find a way around this.

5.4. Low-level Debugging

Any debugger actions which directly reference the values on a datapath or in program variables or which directly display the affect of program operations (other than dataflow actions) within a process will be called low-level debugging. Most low-level debugging will be performed by invoking a separate debugger, which can be any one of a number of standard source-level or symbolic debuggers like adb, sdb, or dbx, chosen and configured into the LGDF2 debugger by the user.

Annotations in this section do not need to attract the user's attention or eye, but serve only as feedback or a reminder that an object is in a certain debugging mode. Simple letter annotations were therefore deemed satisfactory.

By applying the "Print/Debug" menu item from the "Node/Path Action" section to a datapath, the datapath will be annotated with the letter "P". If the datapath state is not neutral, the contents of the datapath will appear on the execution window. As long as the datapath is so annotated, the contents of the datapath will be printed again whenever the datapath state changes to neutral from left or right. The "P" is removed by applying the "Print/Debug" menu item to the datapath again.

Applying the "Print/Debug" menu item to a process causes it to be annotated with the letter "D" (upper case). This annotation gives a slightly different interpretation to "stuck"; when the process becomes stuck, it will be stuck at a low-level-debugger breakpoint within the grant, reserve, or start-execution logic.[2] Along with turning the process temperature to stuck (red or deeply shaded), the low-level debugger will print its standard "stopped at breakpoint" message within the execution window.

When the low-level debugger is invoked in this way, the process cannot be unstuck except by resuming it through the low-level debugger by issuing the appropriate command to it in the execution window. This will turn the process temperature back to frozen (yellow or lightly shaded). Of course, before resuming, the user is free to display the values of variables and/or set new breakpoints within the process. If the "Replay" mode is currently on, the user should not change the values in any of the variables or cause statements within the process to execute out of order.

The LGDF2 debugger does not know about breakpoints which are set by the user with the low-level debugger. Therefore, when these breakpoints are encountered, there will be no feedback in any network window. The only notice of the breakpoint being hit will be from the low-level debugger itself, in the execution window.

As long as the process is annotated with "D", a breakpoint will be tripped whenever the process becomes stuck. The "D" annotation can be removed by applying the "Print/Debug" action to the process again. However, if the low-level debugger has been invoked even once for the process, the "D" will not be removed completely, but

[2]Note that the low-level debugger is not invoked if the process becomes stuck during firing (from being attached to a frozen datapath) since there is no executing process to invoke the debugger on in this case.

instead replaced by a "d". Although this annotation has the same effect as having no "D" annotation at all, it serves as a reminder to the user that, even though the network window may show the process as thawed or frozen, it may indeed be stuck in a breakpoint set by the user via the low-level debugger. The "D" or "d" annotation will be removed automatically when the process suspends.

5.5. Dataflow Tracing

Applying the "Tag/Zap/Untag" menu item to an object will toggle its tracing state between tagged (annotated with a T), zapped (annotated with a Z), and untagged (annotated with neither a Z nor T).

Tagging an object does not really tag the object, but rather the data currently residing in the object. Anytime this data resides within an object with other data, the other data also becomes tagged, and the object carrying the data becomes annotated with a T. Since data can only effect other data if it resides in the same object, this tracing will show how data travels through a network.

Zapping an object really zaps the object - from that point on, all data which resides in the object will be tagged. This is like radioactive dataflow tracing, attributed to Jan Cuny by Nelson and Snyder [5], except that the objects in which the tagged data resides will not in turn become zapped.

The specific rules for propagation of "T" annotations is as follows:

(1) If a process fires and has read access to any datapath that is tagged or zapped, or was itself tagged (zapped) before execution, the process becomes or remains tagged (zapped).

(2) When a datapath becomes tagged or zapped from the user applying the "Tag/Zap/Untag" action to it, any process connected to it by an unfinished arc with read permission will become tagged.

(3) When a process becomes tagged or zapped (either by rule 1 or 2 or from the user applying the "Tag/Zap/Untag" action to it), all datapaths to which the process has write permission and to which the process is connected by unfinished arcs are tagged.

(4) When a process fires as untagged and unzapped (because none of the above rules cause it to be tagged or zapped on firing), all datapaths to which the process has write permission are untagged.

(5) When a process suspends, it becomes untagged (but not unzapped).

Tagging and zapping are included to help trace where data is going or why its affects are not felt in a different part of the program. Though much of this can be ascertained statically from the LGDF2 network and its arc permissions, dataflow tracing with tagging and zapping takes the dynamic effects of granting and reserving arcs in different ways into account.

5.6. Other Menu Actions

Menu selections which have not been described fully are described here.

Applying the Open Window action from the Node/Path menu causes a window to be opened, as described at the beginning of this document. Datapath windows may not be implemented.

The "Display Speed" menu controls the maximum number of events to appear on the screen in any one second. If the parallel program becomes compute-bound, or if all active processes are stuck, events may be spaced further apart than this. The speed of the display limits the speed of the parallel program in most cases so that a user mouse event will not be bypassed by buffered-up program events. Therefore, the "Stop" selection not only stops the display, but the parallel program as well.[3] Note that this is very different than applying a global freeze, since the program stops "trying" to progress in this case. Setting the "Display Speed" to "Hyper" supresses the display of all screen events except for temperature changes, and thus allows the parallel program to execute without waiting for screen updates. When the "Display Speed" is relowered, the entire screen will be updated to reflect the current state of all processes and datapaths.

The "Replay" item on the Global Mode menu represents whether the debugger is in replay mode or not. In replay mode, the debugger reads a trace stream which was created during a non-debugger execution of the program. This trace contains only (1) the firing counts for each process when the network stopped (or blew up) and (2) the order in which "rival" processes executed. Processes are rivals if the LGDF2 model does not dictate in which order they must fire and if differing the firing order could effect the program outcome; it is rare for an LGDF2 program to contain very many such rival processes. In any case, when the debugger is in replay mode, it guarantees that such processes fire in the same order that they did in the original execution to guarantee that the program will produce the same answer and/or encounter the same error. Once the "Replay" item on the Global Mode menu has been deselected, it cannot be reselected, since some rival processes may already have fired out of order.

The Global Action menu can now be explained. Recall that it consists of three items which resemble processes and three which resemble datapaths. The remainder of the markings will depend on whether the current Node/Path Action selection is Freeze/Thaw/Unstick or Tag/Zap/Untag.

- If Freeze/Thaw/Unstick, each of the three process-like items will have a different temperature, as will each of the datapath-like items. Each represents all of the objects on the screen with the like shape and temperature. Applying Freeze/Thaw/Unstick to the red process-like item, for example, will change not only its own temperature to Frozen, but also the temperature of all red processes

[3] In reality, the processes themselves may continue for a short time, until encountering their first dataflow action, but this will not be apparent to the user.

on the screen. After the user has changed the temperature of any of the items in the menu, other items may change temperature so that all three temperatures are still represented.

- If Tag/Zap/Untag, the items in the Global Action menu will not have temperature but rather each will have a different trace state (T, Z, or blank). Except for this, functioning is the same. For example, by applying the Tag/Zap/Untag action twice to the datapath-like item annotated with T, it and all datapaths containing a T on the entire screen will be cleared of tagging (as will all datapaths containing a Z, since the datapath-like item becomes a Z on the first application, then a space on the second).

This definition of the Global Action menu gets a little hazy if Freeze/Thaw/Unstick is implemented as a separate mouse button. In this case, the menu may need another item to toggle between the representations. Future design may allow use of this menu to set global Timed Freeze values as well.

6. Summary

We believe that this design has a number of novel features, including the clean division of high- and low-level debugging, the minimization of recording needed for "Instant replay" debugging, the "temparature" approach (Freeze/Thaw/Unstick) to breakpointing and single-stepping processes at a high level, and the use of data flow debugging (Tag/Zap) to find actual data flow at a high level. However, the only way to truly study the utility of these techniques will be to implement (or at least prototype) the debugger and put it into the hands of users.

Instead of simply adding features blindly, we have attempted to base them soundly on the firm foundation provided by the LGDF2 model. In doing so, misleading concepts such as a "global program state" have been avoided.

7. References

[1] D. C. DiNucci and R. G. Babb II, "Practical Support for Parallel Programming", in *Proc. 21st Hawaii Int. Conf. on System Sciences,* vol. II: *Software Track,* Jan. 1988, pp. 109-118.

[2] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", *Inf. Proc. 74,* North-Holland, 1974, pp 471-475.

[3] T. J. LeBlanc, J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers,* vol. c-36, no. 4, Apr. 1987, pp. 471-482.

[4] D. C. DiNucci, "High-level parallel debugging with LGDF2 (extended abstract)", *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* May. 1988, also available as Oregon Graduate Center Technical Report CSE88-007.

[5] P. A. Nelson, L. Snyder, "Programming Paradigms for Nonshared Memory Parallel Computers", in *The Characteristics of Parallel Algorithms,* MIT Press, 1987, Cambridge, pp. 3-20.