# AmalGUI: A User Interface for a Clustered Programmable-Reconfigurable Processor Simulator

Chi-Wei Wang, Derek B. Gottlieb, Jeffrey J. Cook, Nicholas P. Carter
University of Illinois at Urbana-Champaign
{cwang12, dgottlie, jjcook, npcarter}@crhc.uiuc.edu

***Abstract*— In order for researchers to cope with the design development of increasingly complex processor architectures, architecture simulators must be able to output more information in an efficient and concise manner. AmalGUI is a Tcl/Tk graphical front-end for the Amalgam architecture simulator, AmalSim. Its multiple-window interface provides an effective method of presenting the AmalSim output to the user. Furthermore, it shows how Tcl/Tk can be used to create graphical interfaces for processor architecture simulators and presents a framework for such a tool.**

## I. Introduction

Researchers must be able to keep track of numerous processor states in order to verify proper application execution on increasingly complex processor architectures during design development. Design verification and performance evaluation of a new processor architecture usually requires the creation of a simulator. Such a simulator can provide the user with debugging information on the configuration and the state of the architecture's components as the simulator steps through the execution of application code on the architecture. The amount of state information, though, depends on the complexity of the architecture, which affects the number of components that the user must observe during the execution of an application. Furthermore, this state information is quickly invalidated as a simulator steps to another cycle. As a result, the amount of information and output generated by the simulator can become substantial and will only grow as architectures become more complex.

Text based command-line simulators do not offer a method to output complex state information intuitively and can often impede the user's ability to comprehend the information they provide. They especially lack the ability to efficiently represent complex organizations of components. As a result, there is a need for graphical interfaces that can present this information in an effective manner.

AmalGUI is a Tcl/Tk graphical front-end to AmalSim, our Amalgam architecture simulator, that provides the user with a multi-window environment. Output from AmalSim is presented in the appropriate windows as either text or a graphical representation (image). AmalSim implements high-level models for every major component in the Amalgam architecture in C and runs as a command-line shell. The shell supports commands for printing debugging information and stepping through the execution of an application on Amalgam.

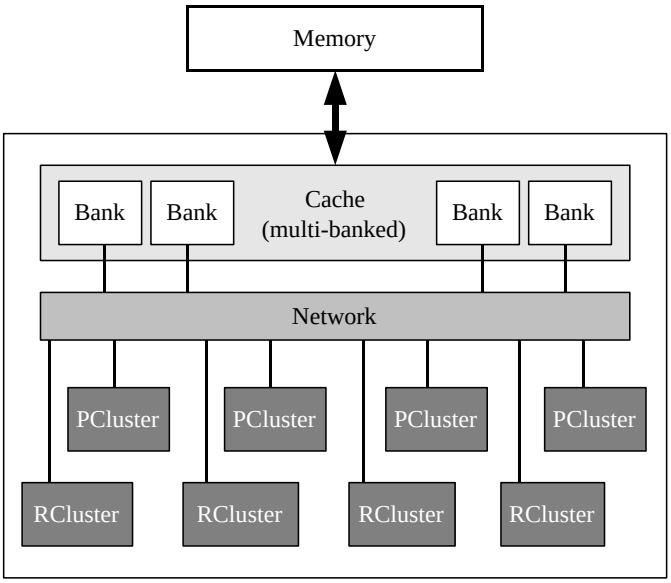



Fig. 1. Amalgam Architecture [1]

## II. Amalgam Architecture

Amalgam is a clustered programmable-reconfigurable processor that integrates multiple memory, programmable, and reconfigurable units onto a single chip, as shown in Figure 1. Each programmable or reconfigurable unit is an independent processing resource known as a cluster. Global communication between clusters and the memory system is handled by a flexible on-chip network. The global memory system consists of a multi-banked cache and the off-chip memory. Every cluster also incorporates a 32-entry register file for local storage and communication with the network.

When executing an application, the application is distributed among the clusters in order to exploit the application's parallelism. Furthermore, the critical regions of the application are implemented in the reconfigurable clusters, which can result in large speedups in performance [2].

However, in order to verify the execution of a distributed application, the user must track the execution on each cluster. This increases the amount of information the user must process since each cluster is independent and has its own register file. In addition, AmalSim's network model is flexible

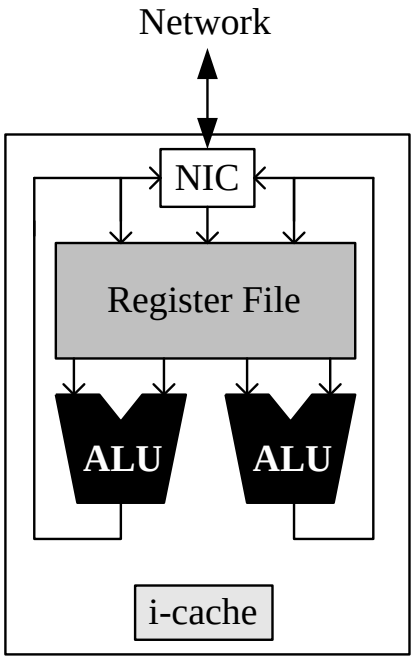



Fig. 2. Programmable Cluster [1]

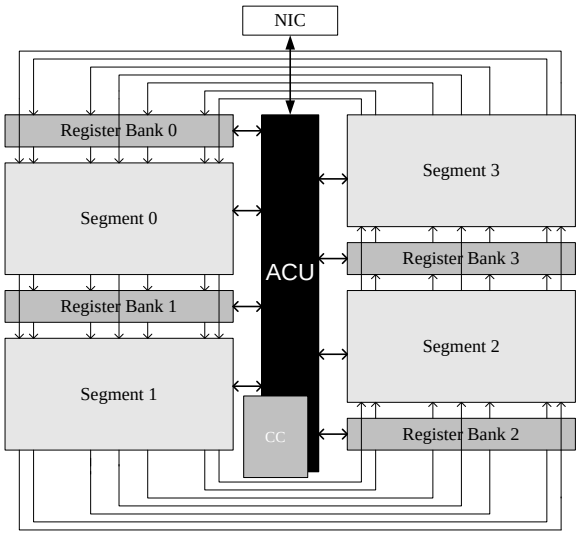



Fig. 3. Reconfigurable Cluster [1]

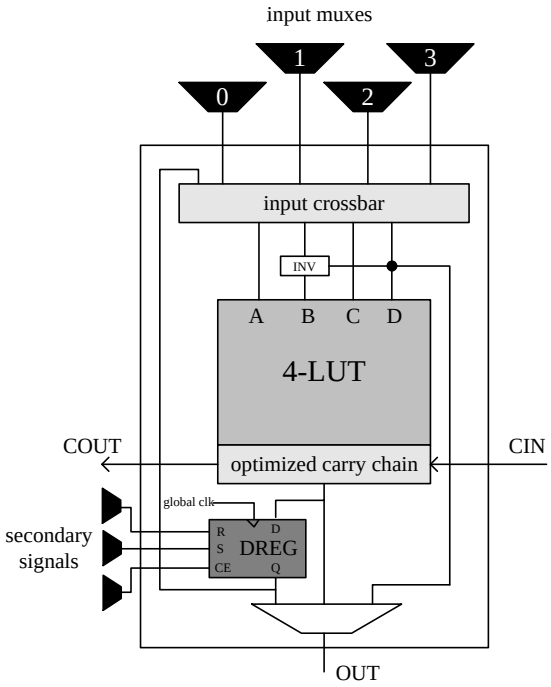



Fig. 4. Logic Block Architecture [1]

enough to offer different configurations that could impact the performance of Amalgam. As a result, the user must also take the network into consideration and observe its behavior during application execution.

Each programmable cluster (PCluster) is a conventional dual-issue in-order microprocessor as shown in Fig. 2. When a programmable cluster executes an instruction, the state of all the components in the programmable cluster may be affected and will need to be observed. The network interface/buffer (NIC) directs incoming and outgoing network data. The dual arithmetic logic units (ALUs) and their pipelines execute instructions and send the results back to the register files. Finally, the instruction cache stores recently executed instructions.

Amalgam's reconfigurable units (RClusters) increase the complexity of the architecture. The architecture of Amalgam's reconfigurable cluster is shown in Fig. 3. At the core of each reconfigurable unit is a reconfigurable array (RA). The RA is a versatile fabric of logic blocks (LB) and wires that can be used to implement multiple circuits during an application's execution. The RA is interleaved with the cluster's register file and then folded to form an unbroken ring that allows computation to flow in a counter-clockwise direction. This reduces wire congestion and wire lengths [1]. Each RA segment has 8 rows and each row has 32 logic blocks. The segment's logic blocks read input data from the 8-entry register bank that proceeds the segment (source) and write to the register bank that follows the segment (destination). For example, in Fig 3 Segment 0 reads data from Register Bank 0 and writes data to Register Bank 1. Activities on the RCluster are directed by the cluster's Array Control Unit (ACU). The ACU also manages the Configuration Cache (CC), a local memory for storing recently used RA configurations.

The logic blocks (Fig. 4) are the fundamental computational units of the reconfigurable cluster. Each logic block can implement any function of its inputs using its look-up table (LUT). The logic block inputs, which the input muxes determine by selecting the appropriate wires, serve as the index of the corresponding output value in the LUT. Around each logic block is a complex and flexible system of wires, which are configured to route data from logic blocks and the source register bank of the segment to the input muxes of other logic blocks and the destination register bank. For each

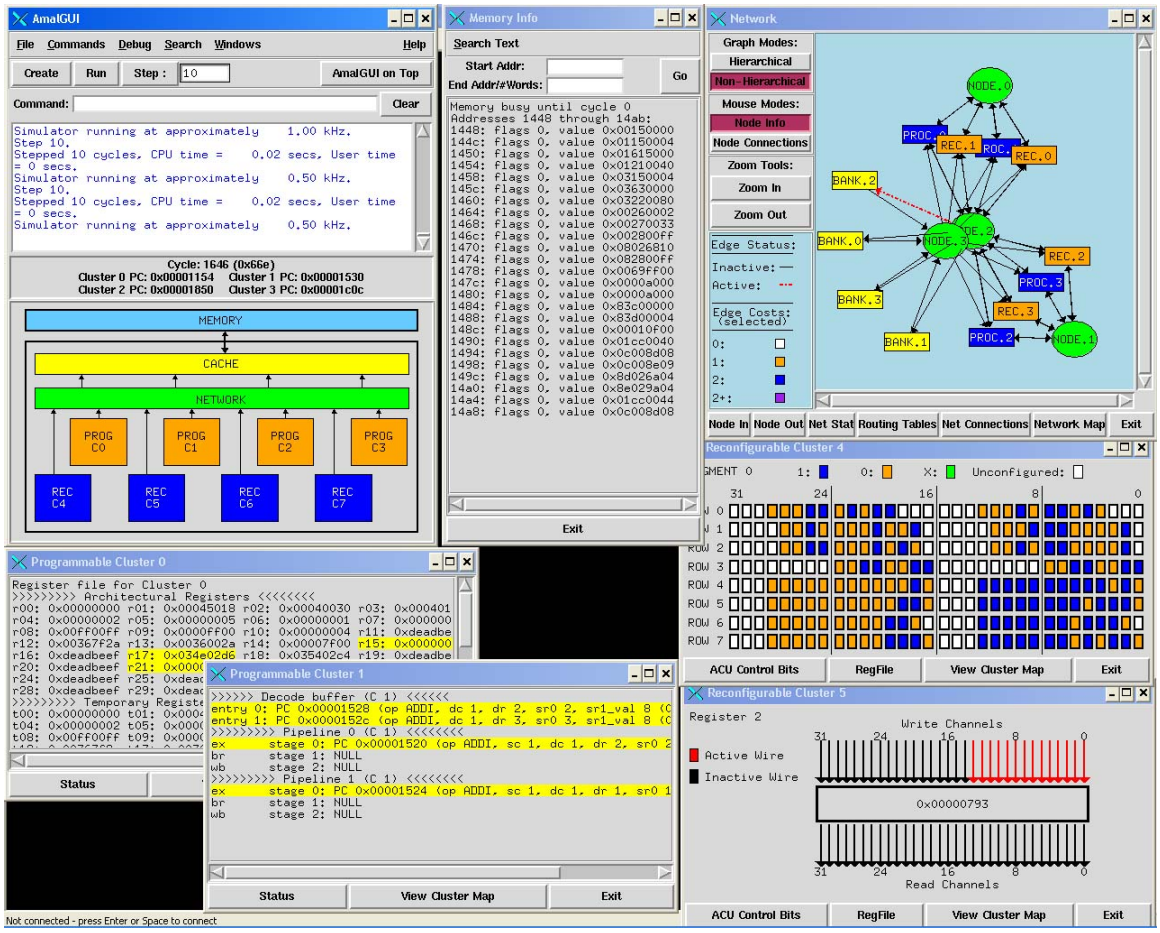



Fig. 5. AmalGUI Screenshot

application, the configuration of the logic blocks (LUT values, input muxes, crossbar, and mode) and the wire routing can vary greatly. In total, each reconfigurable cluster contains 1024 logic blocks and 4096 wire segments. The complexity of the reconfigurable clusters demonstrates the need for an efficient presentation of information to the user.

## III. AmalGUI

AmalGUI displays the output from AmalSim as text or as graphical representations in a multi-window environment. Such an environment lets the user quickly find the information pertaining to a particular cluster or other Amalgam component. Rather than outputting all of the AmalSim output in one window, the information is organized in specific windows that are positioned on the screen according to the user's preference. AmalGUI primarily consists of a main window with multiple component/cluster windows that the user can create (Fig. 5). Each major architectural component in Amalgam has its own window (clusters, network, and memory/cache), although multiple instances of each component window can be created to allow the user to simultaneously query different attributes for the same component.

The main window, shown in Fig. 6, displays important information from AmalSim in a text widget while the cycle and program counter (PC) information is displayed in a label widget. In addition, a canvas widget is used to create a map of the Amalgam architecture.

AmalGUI uses canvas widgets extensively to efficiently show the organization of the architecture and the status of its components. This gives AmalGUI the ability to take large amounts of textual information and present it as a image that is both concise and understandable. The main window's canvas map quickly lets users become familiar with the architecture and its configuration, including how many programmable and reconfigurable clusters are being used by the application running on Amalgam. The map also allows the user to create new windows corresponding to the component the user clicks on. Of these windows, the programmable cluster, reconfigurable cluster, and network windows each use canvas widgets.

**Programmable Cluster Windows:**

The windows for the programmable clusters consist mainly of a text widget, a canvas widget, and various button widgets (Fig. 7). The canvas is again used as a map . Once a component is clicked, the canvas is hidden and the corresponding AmalSim output is sent to the text widget. The buttons can send AmalSim commands that do not have a logical place in the map and also grid or ungrid the map.

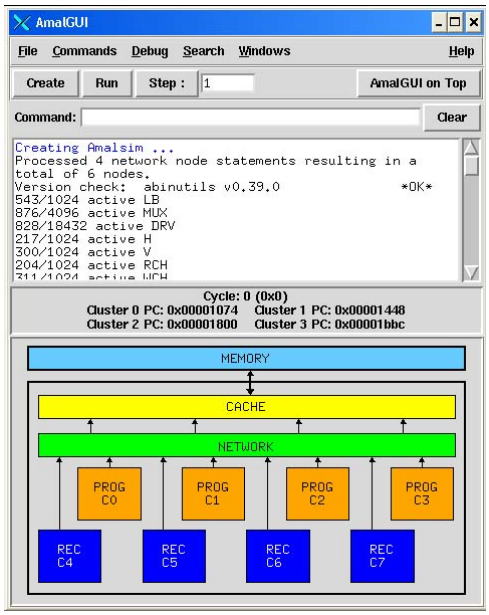



Fig. 6. AmalGUI Main Window

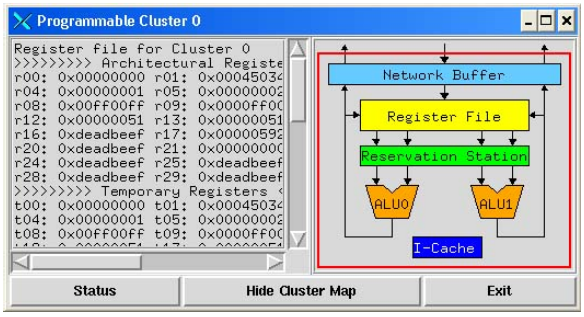
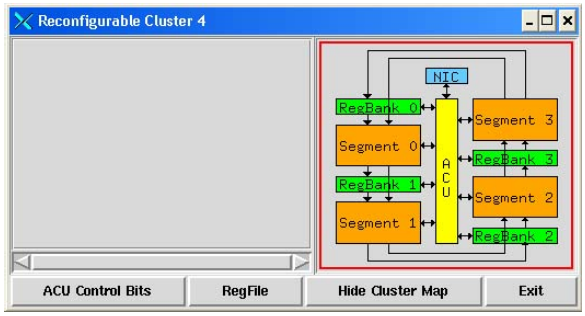



Fig. 7. AmalGUI Programmable Cluster Window

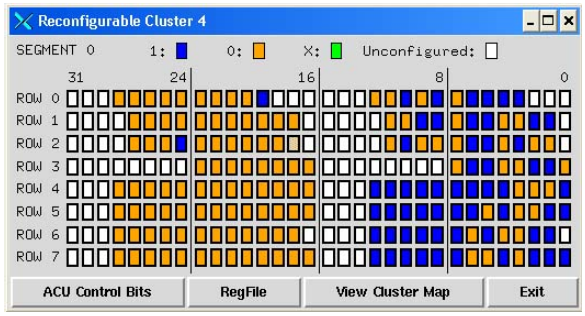
**Reconfigurable Cluster Windows:**

Reconfigurable cluster windows (Fig. 8) are similar to programmable cluster windows. However, an additional canvas widget is often used in place of the default text widget. This second canvas and the graphical representation it creates allow the user to easily and quickly comprehend the large amounts of RCluster information outputted by AmalSim. The canvas is especially powerful when a single image can represent the output of several AmalSim commands.

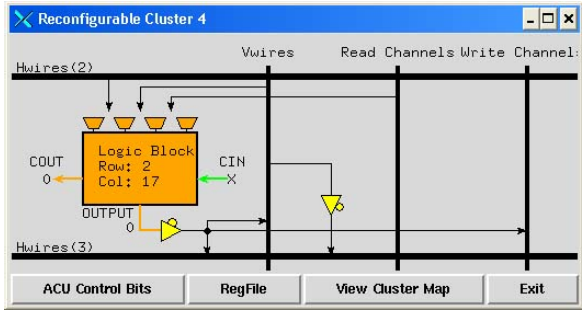
The use of graphical representations produces a reconfigurable cluster window with multiple levels of detail as shown in Fig. 8. Initially, the user is presented with the top most level of the RCluster architecture (Fig. 8(a)). If the user clicks on a component, such as Segment 0, the user goes deeper into the architecture and the canvas is redrawn at a lower and more detailed level as shown in Fig. 8(b), which shows all the LB outputs in Segment 0. If the user continues clicking on a LB, the user will reach the lowest level where details about the LB's configuration and state are displayed (Fig. 8(d)). Bindings to mouse events allow the user to easily traverse between levels to attain the level of detail desired.



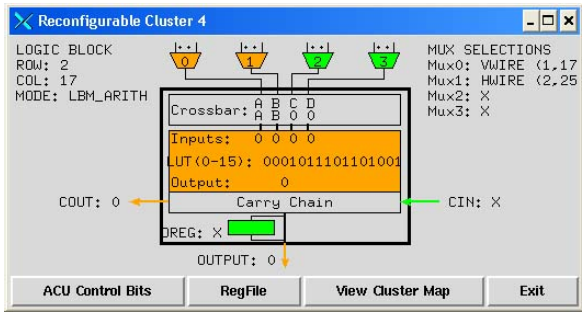

(a) AmalGUI Reconfigurable Cluster 4 Window




(b) View of RCluster 4's Segment 0




(c) View of Segment 0's LB(2,22) and surrounding wires




(d) View of LB(2,22) and detailed information

Fig. 8. Example of multiple levels in RCluster windows.

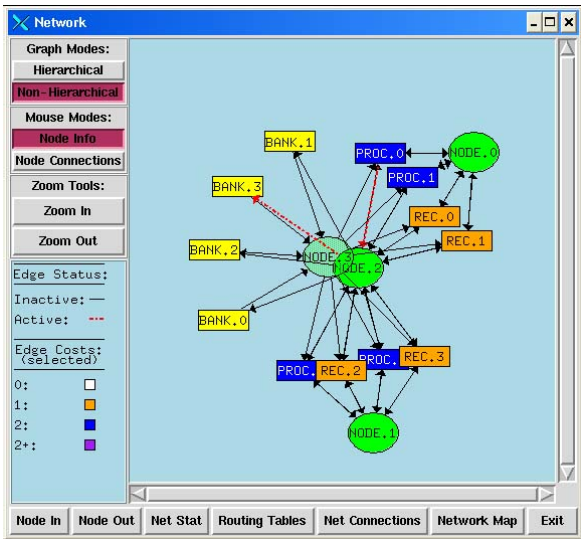



Fig. 9. AmalGUI Network Window

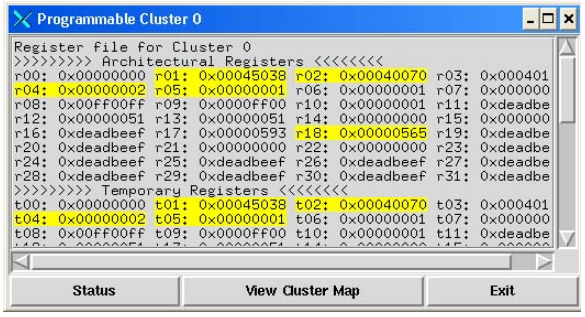



Fig. 10. PCluster Window with highlighting

**Network Windows:**

The network windows use a canvas to represent the configuration of the on-chip network as shown in Fig. 9. All canvas objects are created as a result of bindings with the layout engine in AT&T's GraphViz Tcl extensions. Using the network connection information attained from the output of several AmalSim commands, GraphViz's graph data structure is created. GraphViz then creates a view of how the graph should be displayed by using a incremental layout engine (dynadag, geograph, orthogrid, or fdpgraph) to determine the proper location of graph nodes and edges. Creation, deletion, and changes in the position of a node or edge are bound to Tcl procedures that create, destroy, or move the corresponding canvas object accordingly.

**Updating Windows:**

After a step or run command, the information in each window is updated by re-sending the most recent AmalSim command(s) whose output was sent to each window. If AmalSim output is sent to update a text widget, the new output is compared to the invalidated output stored in the text widget. Differences between the two are highlighted to let the user know that values have changed. Fig. 10 shows a programmable cluster window displaying its register file after an update.

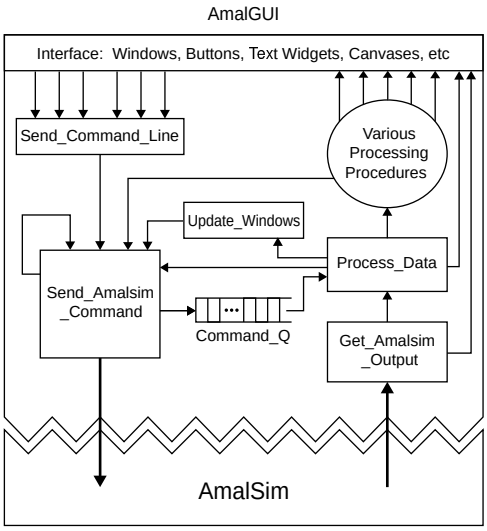



Fig. 11. AmalGUI Architecture

## IV. AMALGUI IMPLEMENTATION

AmalSim is executed as a child process of AmalGUI via the use of two command pipelines, one for stdin/stdout and the other for stderr. This allows AmalSim to be kept as a stand alone command-line shell. The general application architecture of AmalGUI is shown in Fig. 11. The interactions between the user and AmalGUI typically result in AmalGUI sending commands to AmalSim to get the necessary information it needs. These commands are executed by AmalSim, which sends text output back to AmalGUI. AmalGUI then takes the AmalSim output and presents the information back to the user.

**Sending AmalSim Commands:**

The architecture that AmalGUI uses when sending commands to AmalSim can be described as a two-tier structure that allows for efficient checking of a command's arguments. Before commands can be sent to AmalSim, they must have valid arguments and a destination window for their AmalSim output. The arguments are important since they are often used to help AmalGUI direct information to the proper window. The verification of arguments and destination windows, as well as any other preparation needed for AmalSim output processing, are managed by two main procedures: Send_Command_Line and Send_Amalsim_Command (Fig. 11).

These two procedures take advantage of the fact that the commands AmalGUI sends to AmalSim can be classified as either external or internal commands. External commands are any commands with at least one argument specified explicitly by the user. Typically these are commands that the user types in and wants to send to AmalSim through AmalGUI. AmalGUI's main window has an entry widget that allows users to type in commands. However, users can make mistakes when entering commands and as a result, external commands require verification of the command itself and its arguments.

Internal commands are AmalSim commands that AmalGUI generates. All the command's arguments are correct and no verification is required. Generally, when AmalGUI has to send

an AmalSim command to get information on the state of Amalgam, it can generate the proper arguments itself based on the configuration information and the window from which the command was invoked. For example, when a command is invoked from a cluster's window, the cluster number can be determined from the cluster window's name and passed as an argument. The destination window is often the window that invokes the command. It can also be generated by inspecting the command and its arguments. Since internal commands are always correct, they do not need to be verified by Send_Command_Line.

Send_Command_Line performs the verification of external commands and tries to correct command arguments if they are incorrect or missing. Based on the command, the destination window for the command's AmalSim output is also determined. If AmalGUI is unable to determine the correct arguments or destination window for a command, the AmalSim command is not sent and an error message is sent to the main window. Send_Amalsim_Command sets up data structures and performs any other actions needed to process the command output, such as the creation of the destination window. Finally, the command and its arguments are placed in the command pipe and sent to AmalSim (stdin). The command, its arguments, and the destination are then placed into a global queue, Command_Q.

The Command_Q represents the current AmalSim commands that have been sent to AmalSim, but have not yet completed. Since multiple AmalSim commands can be quickly placed into the command pipeline (stdin) and buffered, the queue allows AmalGUI to correctly associate output with the correct commands. This is very useful when a destination window requires the output of multiple commands or when the user decides to queue up subsequent commands while waiting for the completion of the current command.

**Processing AmalSim Output:**

Once an AmalSim command is sent, AmalGUI waits for AmalSim's output to arrive via the command pipe (AmalSim's stdout). By default, the child process' stdout is buffered until it is terminated. However, this would render it useless for our purpose since the two applications must interact with each other. When buffering was disabled though, the output from a single Amalsim command was broken up and sent as multiple lines of output. AmalSim sent a new line to AmalGUI whenever a new line character was outputted. Since a command's output is received one output line at a time, parameters such as the destination window must be determined before the command is sent. Otherwise, redundant computations occur whenever AmalGUI receives a new line of output. Considering the number of lines processed, such an overhead could quickly become significant.

Get_Amalsim_Output is bound to the event of new data arriving in the command pipelines, as well as broken pipes. If the output line is a message from a watchpoint or breakpoint, it is sent to the main window's text widget. Otherwise, it is the output of the command at the head of the Command_Q and is sent to Process_Data, which directs the processing of output lines. Process_Data also is responsible for initiating window updates. Process_Data sends the output line to the appropriate processing procedures based on the sent command, its arguments, and the destination window. The processing procedures, which use regular expressions extensively, send the output line to the destination window's text widget, parse the output line for the data needed to create canvas objects, or set their attributes, such as color. Sometimes, multiple commands need to be sent in order to gather all the data needed to create a canvas. Once all the data is gathered, AmalGUI creates or updates the entire canvas. If the windows need to be updated, Update_Windows is called and sends the most recent AmalSim Command(s) for each window. Such information is stored in a global array with window names as the index. Output lines that are normally sent directly to a window's text widget are instead compared to the existing text lines in the text widget. Using text tags, the new output line is highlighted when it is different from the corresponding text that it replaced.

Once a command's output is finished, a marker is outputted that lets AmalGUI know that the command at the front of Command_Q is done executing and can be removed.

## V. Conclusion

For this application, Tcl/Tk has many advantages. As many have stated before, Tcl/Tk is a powerful language for making user interfaces. It provides most of the necessary data structures and commands for such a task. One of Tcl/Tk's greatest advantages is the number of widget operations and attributes each widget has. This allows AmalGUI to have great control over each widget, which is necessary for creating a successful interface. This is especially true for the canvas widgets, which AmalGUI extensively used. For our approach, Tcl/Tk's support for efficient regular expressions is also very important, since many different lines of AmalSim output are parsed for information. Finally, as AmalGUI's code size grew, Tcl/Tk was still able to run with reasonable speed.

However, by calling AmalSim as a child process and maintaining AmalSim's command-line shell, AmalGUI is dependent on the format of AmalSim's output. Changes in AmalSim's output format could cause AmalGUI to function incorrectly. Another drawback is with double substitution. Many global variables are specific to a particular window, especially the bindings for the buttons, canvas, and mouse events. Since many windows have the same set of variables, the variable names often included the name of the window they are related to in order to distinguish them apart. As a result, when being used as an argument, the name of a variable would often contain the value of another variable (i.e. name of the window). Arrays were sometimes used instead, since the embedded variable, such as a window name, can serve as the index.

A problem with creating many different types of windows is that the code size becomes rather large because each window type is so different. A general approach where one procedure could produce all the windows was initially attempted. However, it was soon realized that there was too much variance

between parameters. This included the names, attributes, and number of widgets, bindings, and window layouts.

Overall, Tcl/Tk was suitable for creating a user interface for processor architecture simulators. AmalGUI helped reduce the architecture and application development time by being effective at organizing large amounts of information in a way that the user can quickly understand.

## REFERENCES


[1] J. D. Walstrom, "The design of the amalgam reconfigurable cluster," Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[2] D. B. Gottlieb, J. J. Cook, J. D. Walstrom, S. Ferrera, C.-W. Wang, and N. P. Carter, "Clustered programmable-reconfigurable processors," in *Proc. of the 1st IEEE International Conference on Field Programmable Technology (FPT)*, Hong Kong, China, Dec. 2002.