VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Graphical Manipulation of Performance Data with Perl/Tk

Steve Wong and Robert Biddle

Technical Report CS-TR-96/10
April 1996

Abstract

Management of complex computer systems is especially difficult because no two systems are completely alike. Accordingly, the performance data essential to management varies from system to system. Moreover, as systems evolve and as management expertise develops, demands on performance data continue to change.

This paper describes a tool for graphically manipulating distributed computer system performance data. The tool allows system managers to define, adjust, and tune performance monitoring information easily, so enabling more effective system administration. The software provides a flexible graphic user interface, yet is capable of manipulating streams of performance data in real-time. The graphic elements are provided by the flexible Tk toolkit, and the fast processing for data manipulation is supported by dynamic compilation using the Perl programming language.

The work described in this paper is part of a larger project, CR, involving development of a performance management system that uses a "building block" approach to allow easily configurable monitoring of heterogeneous distributed systems.

Publishing Information

This was originally published as: Wong, S., Biddle, R., "Graphical Manipulation of Performance Data with Perl/Tk" In *Proceedings of the 13th Uniforum NZ Conference*, Uniforum NZ, 1996.

Author Information

Steve Wong is a Systems Programmer for the Information Technology Services at Victoria University of Wellington. His work involves maintaining Unix and VMS systems, and assisting to expand the services provided the VUW web server. He completed his Master of Computer Science degree at Victoria University of Wellington in 1996. His Masters project involved developing a graphical dynamic manipulator for the "Control Room" project.
Dr. Robert Biddle is a senior lecturer in computer science at Victoria University of Wellington. His main research is in software engineering, particularly in software reusability and in user interface design. He is also interested in distributed systems management, and has worked with Unix since 1974. Robert studied at the University of Waterloo in Canada, and the University of Canterbury in New Zealand. He also has experience in development of commercial computing applications, and in research and development in industry.

Graphical Manipulation of Performance Data with Perl/Tk

Steve Wong and Robert Biddle
Victoria University of Wellington
{Steve.Wong,Robert.Biddle}@vuw.ac.nz

Abstract

Management of complex computer systems is especially difficult because no two systems are completely alike. Accordingly, the performance data essential to management varies from system to system. Moreover, as systems evolve and as management expertise develops, demands on performance data continue to change.

This paper describes a tool for graphically manipulating distributed computer system performance data. The tool allows system managers to define, adjust, and tune performance monitoring information easily, so enabling more effective system administration. The software provides a flexible graphic user interface, yet is capable of manipulating streams of performance data in real-time. The graphic elements are provided by the flexible Tk toolkit, and the fast processing for data manipulation is supported by dynamic compilation using the Perl programming language.

The work described in this paper is part of a larger project, CR, involving development of a performance management system that uses a "building block" approach to allow easily configurable monitoring of heterogeneous distributed systems.

1 Introduction

The Control Room distributed system performance monitoring system (CR) [Zha93, BHZ93] is designed to collate both system and network data to form a visualisation of the performance of a distributed system. In order to present a useful visualisation, any raw data must first be processed by some method to give more meaningful information. In CR, the raw data from nodes is turned into performance metrics via a manipulation process.

Currently, the manipulation is performed by programs written in C. Although the programs process the data with great speed, changing the programs using the standard *write* → *compile* → *execute* → *debug* programming cycle takes a significant amount of time. Moreover, the process of determining how to best extract useful metrics from the data is basically an exploratory process, because one does not know exactly which combination of data sources will provide useful and relevant information.

In an exploratory process, many attempts will be made before coming up with a useful performance metric. When writing C programs to manipulate the data, the time taken to implement a solution becomes problematic, because more time is spent implementing a possible solution than evaluating the usefulness of the resulting output. What we would like is to enable the person manipulating the raw data to spend less time implementing possible solutions, allowing more time for exploration and interpretation.

The complicating problem is that it is still necessary for the data to be processed with great speed. The solution we have implemented uses Perl (a byte-compiled language) to speed up the implementation cycle, and the Tk extension to Perl to provide a dynamic data-flow oriented manipulator, with an interactive graphical user interface. This paper describes a prototype for dynamically manipulating raw distributed system: NCM, the **N**ew **C**ontrol Room **M**anipulator.

2 Background

2.1 CR Architecture

The CR architecture design is based around the need to monitor the different parts of a distributed system, but in a generalised way. To this end, it has to be able to monitor not only Unix based machines,

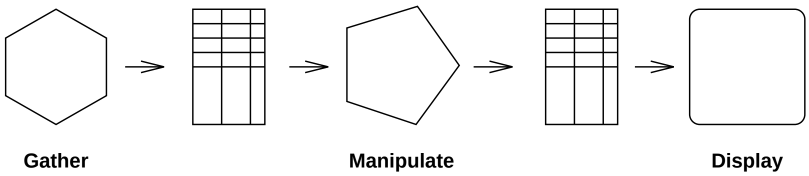

Figure 1: The CR architecture.

but other operating systems as well. Additionally, CR is not just used to monitor host computers, but other nodes in a distributed system as well, such as routers and bridges.

The CR framework is made up of three main components, which are linked via two shared memory databases, as shown in figure 1:

- **The Data Gatherers** collect the data from the different parts of the system. Data gathers put collected data into a shared memory segment, thus making the shared memory segment a data collection point for the system.
- **The Data Manipulator** takes the data put into the shared memory segment from the gatherers and processes it to form useful metrics which are them placed in the other shared memory segment. The metrics can then be presented by the display units to the user.
- **The Data Display** is important as we wish people to be able to monitor the data. The display units read the metrics from the second memory segment and then display it in a human readable from. As the amount of data is quite large, the display is configurable by the user to provide as much or as little detail as they want.

2.2 The Role of Manipulator: Usability and Performance

In the CR architecture, the data manipulator sits between the two shared memory databases, performing transformations on the data. The data that is collected by the gatherers (*raw data*) is placed in the *Raw Metric Base* (RMB). Data is taken from the RMB by the manipulator, and than turned into *cooked data*, which is placed in the *Performance Information Base* (PFIB).

The raw data that is collected by the gatherers is often not useful without some form of processing to turn it into *cooked data*. The manipulator performs the transformations to change the raw data into the cooked data for the CR system. An example might be that we want to monitor the number of undergraduate students logged on the system at any one time. To do this, each host would have a gatherer process which reported the number of undergraduates logged on at a particular time for each host. To find the total number of undergraduates logged on, the data reported from each host must be processed to determine the number of students actually logged on.

Without the manipulator, this simple transformation could logically be performed by the data gathering component, but this would result in the following problems:

- It is difficult for the data gatherers to perform the manipulation, as they cannot communicate with the other gatherers.
- If it were possible for the data gatherers to communicate, each one would perform the same transformation on the data, resulting in several processes performing exactly the same task.

These problems are solved by the use of a centralised processing component, the *data manipulator*. The data manipulator is able to read the raw data from the RMB, perform all the processing, and store the cooked data in the PFIB for access by both the display and logging components. In addition, as the manipulator is a central processing point, it is able to maintain data integrity, and is able to re-use intermediate calculations.

The manipulator provides a central processing point to transform raw data flowing into the CR system into useful metrics before they are displayed, so that we may monitor the state of the system. Because of the wide number of variables in a distributed system, it is difficult to determine a set of useful metrics. Because of the trail nature of the new metrics, the manipulator should make it easy to specify transformations, in order to reduce the time used in the write and debug phases of the exploration cycle.

Ideally, the manipulator should be able to process all the manipulations as fast as raw data arrives into the shared memory segment. This aim is complicated by the following:

- Gathering processes have very little computation to perform before the raw data gets put into the shared memory segment. It will always be more plausible that the data gatherers are more likely to generate data faster than the manipulator can process it.
- Incoming raw data is placed in the shared memory segment by the gatherers at different times, so updating as fast as the data comes in means that the manipulations should ideally be performed every time the relevant raw data changes state.
- The manipulator also has to perform many calculations before it can finish processing a given set of metrics. On top of this, the manipulator also has to support a user interface, which generally tends to cause a heavy toll on performance.

The feasibility of processing data as fast as the data is generated is directly related to the number of raw data updates that are made per second by the gatherers. When deciding how many times a second a gatherer should update, we must consider how much information is useful, traded off against the processing required to get the data. To allow the gatherer to update as fast as possible is not an ideal situation as it impacts upon system performance, so a balance must be reached so that there is as little effect on system performance as possible.

Also important is how much information is actually useful for monitoring purposes, especially in the case of logging. If there are many updates per second, then log files will quickly become unnecessarily large. However, it may be useful to have highly detailed log data in order to analyse system performance in critical or exceptional situations. For simply monitoring data via a display, the data does not need to be sampled as often, as there are limits to both the human perception system, and the ability of a human to process information.

The requirements must really reflect reality for distributed systems. As an rough guide, for example, we would consider systems of 100 nodes realistic, and would want to look at about 10 performance figures for each. And we need to monitor such systems continuously. Ideally, the manipulator would be able to process 10 metrics for each of 100 nodes in one second. As a minimum, a manipulator is expected to be efficient enough to process 10 metrics for each of 100 nodes in five seconds.

Lastly, the manipulator should also be able to react to give quick response to user interaction, as it is supposed to be an interactive application. If the manipulator is too slow to use, then the write-debug cycle becomes longer.

2.3 The Graphical Data Flow Approach to Programming

CR provides a system-wide visualisation of a live distributed computing environment, so that a better understanding may be achieved of a complex system. CR also provides a monitoring system for the analysis of system performance and the early detection of problems. The dominant high-level theme in CR is the *flow* of data from points around the system, and through the CR components.

This idea of information flow has been considered in other contexts, and has proven a useful basis for visualising systems and controlling software. The data flow model has been especially significant in the design of visual languages. Such languages encompass general computing, as in Prograph [CGP89], or are more specific in their intent. LabVIEW [Nat91], for example, is a data flow visual programming language aimed at controlling laboratory equipment.

In the data flow model of computation, a program is executed by the flow of data within it, rather than the flow of control used in the conventional von Neumann model of computation [Shu88]. In the model, a data flow program can be represented by a directed graph, with the nodes of the graph representing operations, and the edges representing the data flow between nodes.

Nodes execute their operations as their inputs become available, instead of when control is passed to them, so there is no control structure other than that which is implied by the data dependencies of the nodes. In the normal data flow model, nodes consume their inputs and produce outputs, but only when all their inputs are available.

Data flow languages lend themselves to graphical representations because of the following [DK82]:

1. Data flow languages sequence program actions by a simple data availability firing rule: When a node's arguments are available, it is said to be "fire-able." After firing, the node's results are sent to other functions, which need the results as their arguments. This behaviour suggests that the

program can be aptly portrayed as a directed graph in which each node represents a function and each directed arc a conceptual medium over which data items flow.

2. Data flow programs are easily combinable into larger data-flow programs.

3. Graphs can be used to present an intuitive view of the potential concurrency in the execution of the program.

4. Graphs can be used to attribute a formal meaning to a program.

All this suggests that the graphical data flow model should fit well with the needs of CR. The graphical and dynamic nature should support the exploratory use of CR manipulator, and the structures for general programming should support any complexities that may arise.

3 NCM

3.1 Outline

Previously, it was stated that the time taken to implement a metric took too long with the traditional *write* → *compile* → *execute* → *debug* cycle. Thus it was proposed to develop a graphical language that was based on a data-flow approach to form the basis for a new manipulator. The new manipulator would use the approach demonstrated by visual data flow languages and allow the user to interact with the manipulation subsystem, similar to the way LabVIEW allows users to interact with the physical systems in a laboratory.

3.2 Technology

As the manipulator is to be able to dynamically modify the way data is transformed from the raw state to its cooked state, it would be advantageous to implement the manipulator in a language that did not require the application to be restarted once a change was made.

This discounts C and other compiled binary languages, as changing the way the program behaves would require the manipulator to be stopped, recompiled, and then restarted. Therefore we turn to interpreted languages, whose programs can modify themselves whilst they are running, and so do not have to be shut down and restarted to change their behaviour.

An obvious choice for the implementation language would be TCL/Tk [Ous94], as the Data Display is also implemented with this. TCL unfortunately has some shortcomings if we were to use it to implement the manipulator:

- TCL is a purely interpreted language, so code is **re-evaluated** every time it is called. Because of this, TCL code is too slow for our purposes.

- The TCL language does not efficiently support complicated programming, especially numerical computation — TCL users typically supplement it with compiled C for this kind of functionality.

One of the main advantages of using TCL/Tk would have been the ability to use Tk to implement the user interface, as the Tk toolkit is both sophisticated and powerful, whilst still retaining a great deal of simplicity.

However Perl/Tk (`pTk`) combines Perl [WS91] and Tk. Unlike TCL, Perl does not re-interpret code as it is used, rather it byte-compiles code as much as possible before execution. When code is modified during execution of a program, the modified code is also byte-compiled as much as possible. This means that each piece of code only gets interpreted once, when it is first seen by the interpreter. This means that although the code is not executed as fast as a C program, it is still executed much faster than it would if it were be to be re-interpreted every time the code is used.

The `pTk` hybrid takes advantage of the object-oriented capabilities of Perl, and also provides helpful extensions to Tk, such as composite widgets. This hybrid seems to provide exactly what is needed for a new CR manipulator: fast execution with good user-interface support. There is one drawback. Currently, only basic documentation is available for `pTk`, and more advanced features are under-documented, if at all. Despite this, we chose `pTk` for this project, and persevered. We expect better documentation will soon become available.

3.3 Design Overview

The manipulator is basically split into two parts which perform the different functions which are required. One part is used to represent the different entities which are required for the execution of the manipulations. The other part is used for the user interaction to specify the nature of the manipulations.

In NCM, a graph is represented in two ways simultaneously. A user draws their graph on the user interface, and has a visual representation of their nodes on their screen. Each node in a users graph can be thought of as an invocation of a small Perl *subprogram*, which takes inputs and produces outputs. Internally to NCM, the information to execute each subprogram is stored in an object, which contains data about the inputs, outputs and Tk widgets required by the subprogram.

When a node is activated, this results in an invocation of a subprogram. The information for each invocation is represented by an object, which stores the name of the subprogram to be called, which nodes the outputs go to, and which Tk widgets will be used by the program.

When NCM "runs" a program, it performs a *cycle* on all the nodes in the program graph. A cycle is a single execution of every node on the graph. Node operations in a cycle must be executed in an order that reflects the control flow implied by the data dependencies, otherwise the program will be incorrect. It is simple to automatically determine the correct ordering for a set of nodes, as long as the nodes do not involve themselves in a loop of data dependencies. We have also experimented with other approaches that do allow simple loops.

Because Perl is a conventional control-flow oriented language, the manipulator simply takes ordinary Perl, code and provides the appropriate code to wrap each subprogram used. Each of these invocation calls of those subprograms is also wrapped, so the code can be run in a data-flow fashion. Once this pre-processing is done, execution of the code becomes straight-forward.

The pre-processing stage begins with a list of nodes that are to form the user's program. This list has been ordered, dictating the order in which the invocations must be called. The pre-processor examines the objects associated with the invocation order list and determines which subprograms will be needed, and then proceeds to generate wrappers for the subprogram code.

As a small example, consider the small sub-program below, which would implement a simple formula evaluation node. Like most NCM sub-programs, this takes input from a queue (`in`), and produces output on a queue (`out`). This sub-program assigns the input value to a variable `x`, and takes a formula input by the user in an entry box widget (`entry->get`). The idea is that the expression may include references to `x`, so evaluation will produce a value determined by `x` and the formula. The expression is evaluated and assigned to a variable `y`, which is then pushed onto the output queue.

```
# code for formula node
my $x;
my $y;
$x = shift(@$in);
eval $entry->get;
push(@$out,$y);
```

A formula sub-program node type is built-in to the manipulator, but the user may also create sub-programs of their own. As shown in the example above, the user need only be concerned with code that actually involves manipulating data. The pre-processor generates wrapper code for any sub-program that takes care of necessary headers, declarations, and data scope considerations. This wrapper code also includes any connections necessary between the subprogram and any Tk widgets that associated with the node.

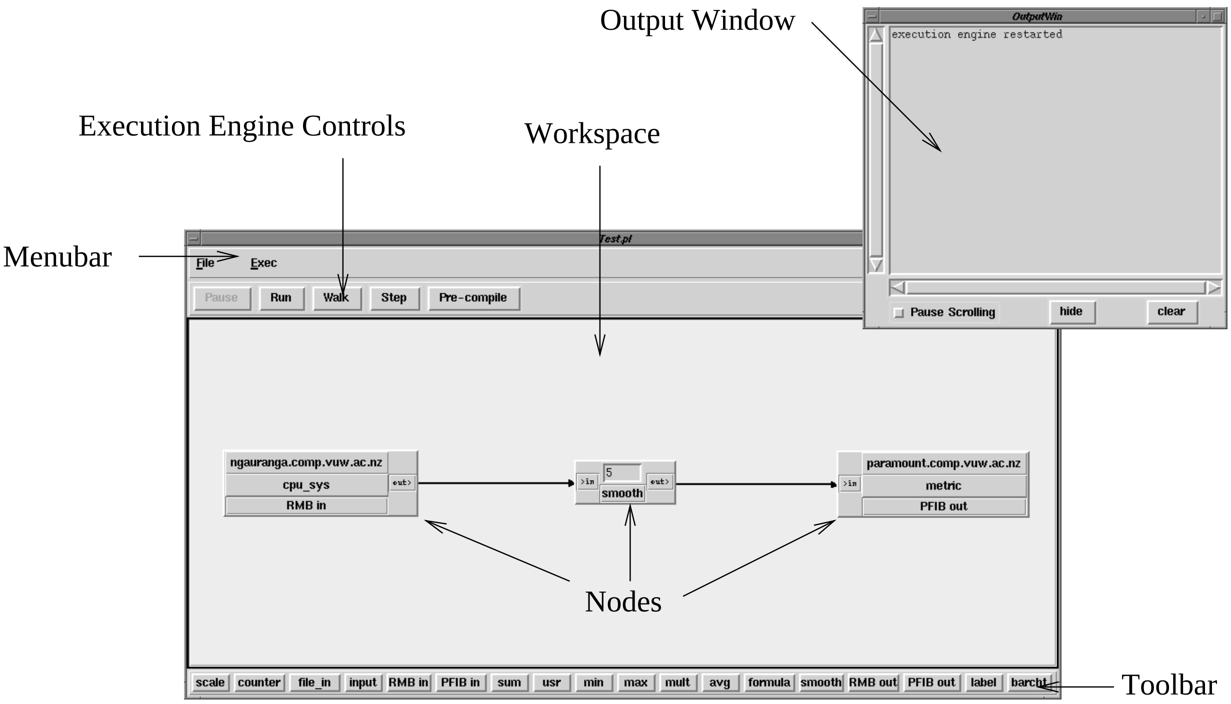

Figure 2: The NCM interface

4 Using NCM

4.1 User Interface

The GUI interface to NCM primarily consists of the *Workspace*, where the users create their programs. Users create programs by dragging different node types from the *Toolbar* (which is located at the bottom of the screen), onto the workspace. Once a user has set up their nodes and connected them together, they can then get the execution engine to run the program.

For easy access, the most commonly used functions for the execution engine have been placed on the *Execution Engine Controls* bar. From here, the user can pre-compile their code so it is ready for execution, and then run the program in a number of ways. Additional execution engine controls can also be found in the *Exec* menu on the *Menubar*, where the user has access to the various utility functions of NCM.

Both the execution engine and the interface itself provide additional feedback to the user via the *Output Window*, for both informational and error feedback to the user. Each *node* in NCM also recognises a right-mouse button click and provides additional control over the attributes of the node. If many of the attributes for a particular node are being changed, then the user can also tear off the right-mouse-button menu to form a floating toolbar for that node.

4.2 Creating A New Component

This example demonstrates how to create a user defined component from scratch. User defined components are provided to enable the user to access Perl language in the event that they wish to make custom node types. The example below details step-by-step creation of a summation sub-program.

1. Drag a *usr* node into the workspace from the toolbar.

2. Right click the **right mouse button** on the label of the node (where it says "usr"). This brings up the *node menu*.

3. From the node menu, select *Edit Title*, and change the value to "my func".

4. From the node menu, select *Add Input...*, and name the new input "in".

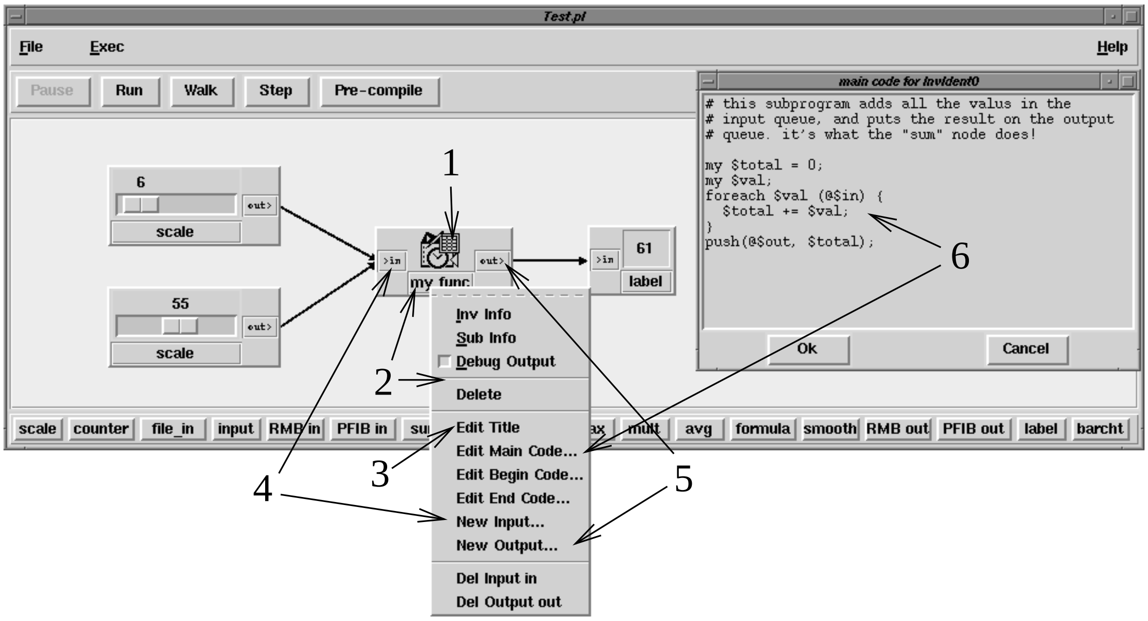

Figure 3: Example of creating a new component

5. From the node menu, select *Add Output...*, and name the new output "out".
6. From the node menu, select *Edit Main Code...*, and enter the following:

```
my $total = 0;
my $val;
foreach $val (@$in) {
  $total += $val;
}
push(@$out, $total);
```

7. Drag a *scale* node into the top-left corner of the workspace.
8. Drag a *scale* node into the bottom-left corner of the workspace.
9. Connect the "out" ports of each scale node to the "in" port for your new node.
10. Drag a *label node* into the workspace.
11. Connect the "out" port of your new node to the "in" port of the label node.
12. Pre-compile the code by pressing the *Pre-compile* button.
13. Execute program by pressing the *Run* button.
14. Press "Pause" once you have finished.

4.3 Built-in Sub-Program Components

Each of the built-in components supply the necessary information to enable the execution of the corresponding subprogram represented by the node. These are the different classes of built-in components:

input	outputs the contents the user has specified in the entry widget of the node	*label*	prints an input to the display label of the node
scale	outputs the value on the scale of the node	*counter*	outputs a continuously increasing number
file_in	given a file, will read a value from the specified file every time it is invoked, and then use the value as output	*barcht*	prints input values as a barchart
min	outputs the minimum value of all the input values received	*max*	outputs the maximum value of all the input values received
sum	outputs the value of the sum of its inputs	*mult*	outputs the value of all the input values multiplied together
avg	computes the average of all the inputs received	*smooth*	smooths the input value by the user-specified smooth factor
RMB in	outputs the value of the given metric from the RMB	*PFIB in*	outputs the value of the given metric from the PFIB
RMB out	writes a value to the RMB segment	*PFIB out*	writes a value to the PFIB segment
formula	applies the function, which the user gives via the entry widget, to the input	*usr*	user-defined node type

4.4 Integrated Example

This example demonstrates how the manipulator is used as part of the CR system. The following graph calculates the total disk usage for one of the nodes in a system. This program also smoothes the data values for the system CPU usage for another node, as well as being able to see the effect of the smoothing with the original data.

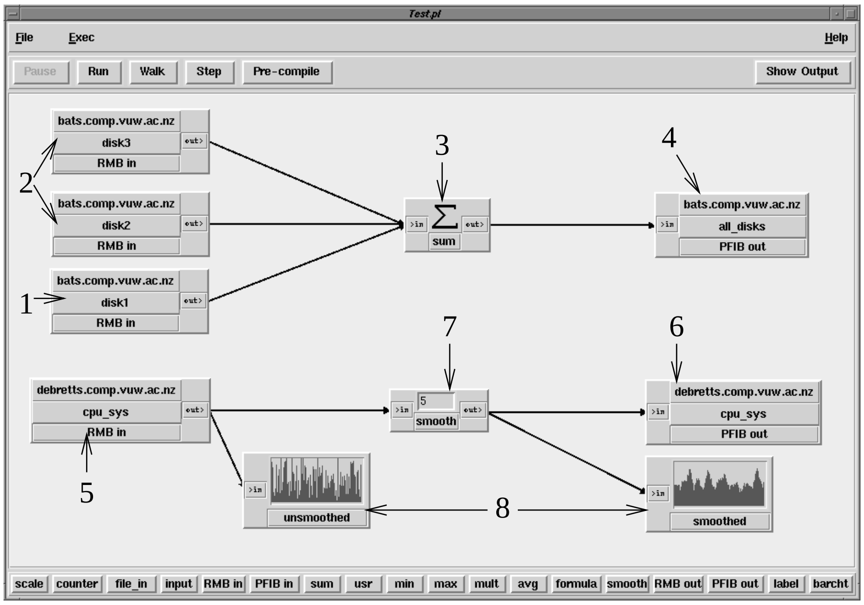

Figure 4: NCM integrated example demonstration.

1. Place an *RMBin* node onto the canvas. Position the mouse cursor above the "node" pull-down menu button, and hold the mouse button down. A menu will appear with the names of various nodes on it. Select "bats.comp.vuw.ac.nz". Do the same with the "metric" menu button, but this time it will come up with the names of various metrics on the menu, so select "disk1". Each time

this node is invoked, it will get the raw disk space used from the RMB shared memory segment, which was put there by a CR gatherer.

2. Place two more RMBin nodes into the workspace. Select (bats.comp.vuw.ac.nz, disk2), and (bats.comp.vuw.ac.nz, disk3).

3. Drag a *sum* node into the middle of the workspace, and connect the data values from the three disk readers.

4. Drag a *PFIBout* node to the right of the sum node. Select the "all_disks" metric for bats.comp.vuw.ac.nz. Connect the input of this node to the output from the sum node. The PFIB metric will now contain the usage of all the disks.

5. Drag another RMBin node into the workspace, and select the metric (debretts.comp.vuw.ac.nz, cpu_sys).

6. Drag a PFIBout node into the right hand side of the workspace, and select the cpu_sys for debretts.comp.vuw.ac.nz.

7. Right now this metric isn't going to be very interesting, but what we really want to do is smooth the value, so drag a smooth node into the workspace between the raw and cooked nodes, and then connect them together.

8. We want to be able to see the effect of the smoothing function, so we should look at the values before and after they pass through the node. To do this, we can add two *barchart* nodes to the graph. One should be connected to the raw cpu_sys node, and the other to the output of the smooth node. (It would probably be useful to make the graphs longer, and to give the nodes useful titles. This can be done via the right-mouse-button menu.)

9. Run the program. Once the manipulator begins running the program, you can check the metrics on a CR display.

5 Evaluation

5.1 Usability

One problem not tackled in this prototype concerns large graphs that are too big to fit onto the workspace, or those which become unmanageable for the amount of room available. Even though users can bunch groups of nodes together to order to signify they co-operate in performing some set task, some graphs will still be too big to fit comfortably on the workspace. So while we have spatial organisation, another mechanism is required to help organise large graphs. One tool that might help is a zoom function (currently not provided in Tk), which would allow the user to view more of the program in the same area. Another alternative would be to allow some sort of grouping mechanism in which a group of nodes could be shrunk into one 'virtual node', therefore breaking the graph into manageable chunks.

Related to the previous problem is the problem of repeated tasks. There is currently no mechanism for repeating commonly used sets of operations. For example, imagine a metric which measures the I/O activity against CPU activity, and this metric was found to be useful. Presumably the metric would have been implemented on just one or two machines. Quite obviously, we would now like to use the same set of nodes to calculate the same metric for all other hosts without repeating the nodes. Currently, there is no facility to do this, and will rapidly become a fast-requested feature, especially as there is no way to copy sections of a graph as yet!

Resolving these problems was outside the scope of this prototype project, and will be addressed in further development. Solutions have been developed for similar problems in other visual data flow languages, and should be generally applicable.

Despite the problems, the combination of the data-flow model and the Perl control-flow oriented language to implement node subprograms works well. The data-flow language is able to simplify the general way nodes are invoked, and in what order, where a control-flow language would make it look complicated.

At the same time, Perl provides a powerful computation base and a familiar general purpose language that is familiar to system programmers. It therefore provides an excellent way to specify calculations where often many nodes would be required in a data-flow language to say the same thing. Many data-flow programs have been known to look awkward because the programmer was trying to implement a control-flow solution in a data-flow language.

5.2 Performance

The performance for the manipulator is both dependent on the rate the incoming raw data is to be sampled, and the amount of spare computation that is required to run the interface.

Aside from the noticeable slowdown with managing the user interface, there are other areas in which performance could be increased. The code generation system is quite naive and many optimisations could be made to the generated code, such as:

- Variable assignment optimisation during context switches to remove unnecessary copying of variables.
- Optimise invocation call order so that variable optimisation has more effect.
- Optimise calling of nodes so that nodes are not invoked when they do not need to be called.

Measuring performance within Perl is difficult because there is no easy way to access the time in milliseconds. This means to get accurate timing, test cycles tend have to be of around 10000 cycles in order to ensure some degree of accuracy.

In order to see how long it takes to execute an "ordinary" node, several test trials were run at 5am on a Sunday morning in the hope that little other activity would be occurring on the test machine. The tests ran a graph consisting of a number of nodes, for 10000 cycles. Because the nodes which involve Tk widgets can be expected to take longer, there are two sets of tests, one for sum nodes, the most minimal node type which does no Tk updates, and the counter and label nodes, each of which performs a very simple update.

The tests were executed with an inter-cycle run delay of zero milliseconds, the results are shown in table 1 and figure 5:

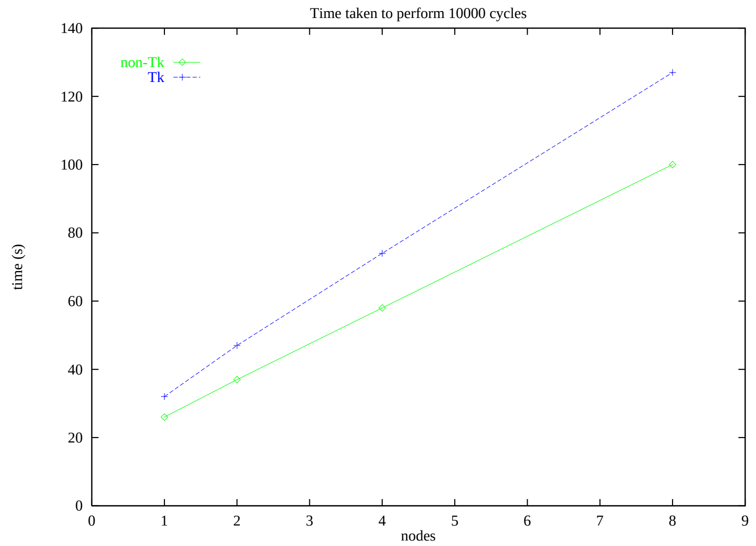

Figure 5: Graph of benchmark trials.

From these results, one can see that each additional non-Tk node that we add to a program after the first node adds approximately 10.57 seconds to a benchmark run of 10000 cycles. There appears to be a total overhead of about 15 seconds for running of the rest of the manipulator during the benchmark. For Tk nodes, one node will add approximately 13.57 seconds per 10000 cycles, and there appears to be a total overhead of about 18 seconds.

The results suggest that the manipulator could execute a single non-Tk node in approximately 1.06 milliseconds at best with a manipulator overhead of 1.54 milliseconds per cycle, and a Tk node in

Type	1 node	2 nodes	4 nodes	8 nodes
non Tk	26s	37s	58s	100s
Tk	32s	47s	74s	127s

Table 1: Time taken to run 10000 cycles

approximately 1.36 milliseconds at best, with an overhead of 1.84 milliseconds per cycle. An initial reaction might be to suggest that a node which uses a Tk widget only has a 0.3ms processing disadvantage, but this is due to screen updates being cached by Tk. That aside, we can estimate the best-case performance to be the following shown in figure 6.

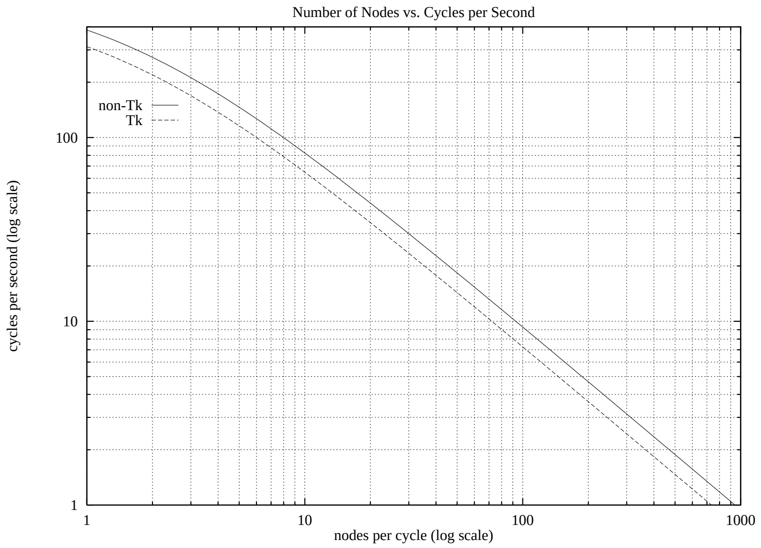

Figure 6: Best-case performance for NCM

The data shows that NCM can process around 900 node invocations per second. The minimum performance requirement for the manipulator is the processing of 10 metrics for 100 nodes in 5 seconds, which is 1000 metrics in 5 seconds, or 200 metrics a second. This means that the manipulator can allocate 4.5 nodes per metric and still fulfill the minimum performance requirement, but can only allocate 0.9 nodes per metric when trying to fulfil the desired performance requirement.

It was hypothesised that the use of queues for passing around the data values was slowing down the manipulator. So it was proposed that the manipulator could be speeded up by replacing the inputs and outputs of nodes from pointers to arrays to ordinary scalar variables in the Perl code. After writing a small Perl program to test how fast scalar assignment-fetch pairs were against array reference enqueue-pairs, the following results were obtained:

Type	Time for $500,000$ Accesses	Average Time
scalar assign-fetch pairs	43 seconds	0.086 ms
array reference enqueue-dequeue pairs	79 seconds	0.158 ms

Table 2: Comparison of the speed of scalar variables against array references

If we assume that all nodes use a single enqueue-dequeue pair to move a single value through the node, ie. take a value from an input queue, and put a value onto an output queue, then we can assume that we can can save $(0.158ms - 0.086ms)$ per value moved per node invocation. This represents an estimated saving of around seven percent in node execution time. This suggests that it may be useful to add type information to the inputs and outputs of subprograms.

6 Conclusions

The aim of this project was to produce a new manipulator for the CR project that could dynamically modify the manipulations performed for the CR system, and to improve the way the manipulations were written. NCM is only a prototype, but it has successfully demonstrated both design principles and implementation technology that hold much promise.

The requirements for the new manipulator involved a combination of high usability and high performance. We have addressed the usability requirement by fostering exploration in a graphical data-flow programming language. This does seem to provide a good conceptual model to the user, although solutions have to be found to the problems of handling large graphs and the reuse of sections of programs.

We have addressed the performance requirement by using a byte-compiled language, and performance measurements indicate this is a reasonable approach. NCM can perform around 900 node invocations per second, which fulfils the minimum performance requirements, and the internal NCM engine can yet be optimised further. Moreover, the combination of an efficient byte-compiled language with a powerful user-interface toolkit seems to have been especially useful in allowing us to address both usability and performance requirements together.

References

[BHZ93] Robert Biddle, John Hine, and Zhiqing Zhang. CR: A Monitor for Distributed Systems. In *Proceedings of the Uniforum NZ Conference*, 1993.

[CGP89] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual co nditioning. In *IEEE Workshop on Visual Languages*, 1989.

[DK82] A. L. Davis and R. M. Keller. Data Flow Program Graphs. In *IEEE Computer*, volume 15, pages 26–41, February 1982.

[Nat91] National Instruments Corporation. *LabVIEW 2 User Manual*, September 1991.

[Ous94] John Ousterhout. *TCL and the TK Toolkit*. Addison-Wesley, 1994.

[Shu88] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, Inc., 1988.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly, 1991.

[Zha93] Zhiqing Zhang. CR: A Monitor for Distributed Systems. Master's thesis, Victoria University of Wellington, 1993.

Version Information

- Perl : the current version of `perl` is `5.002g`, however the `perltk` this project uses is `5.001n`.
- `pTk` : the current version of `pTk` is `b10`, which is a *beta* release of the extension, and uses version `4.0b2` of the Tk widget set. The manipulator currently uses the `b8` version of the `pTk` extension, which still has some minor bugs. `pTk` is currently being maintained by Nick Ing-Simmons (email: `nik@tiuk.ti.com`).