Runtime Visualization of Computer Architecture Simulations

H.C. Kok, A.D. Pimentel, L.O. Hertzberger

Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{kok,andy,bob}@wins.uva.nl

Abstract

The discrete–event simulation language Pearl, which is specifically designed for simulating computer architectures, features a strong statistical analysis engine. However, the output of this engine is text–based and postmortem. In this paper, we introduce a flexible graphical user interface support library for Pearl, addressing the runtime visualization of computer architecture simulations. The hierarchial structure of the library is highlighted and, with the help of a case study that was extracted from previous work, the merits of the library are presented.

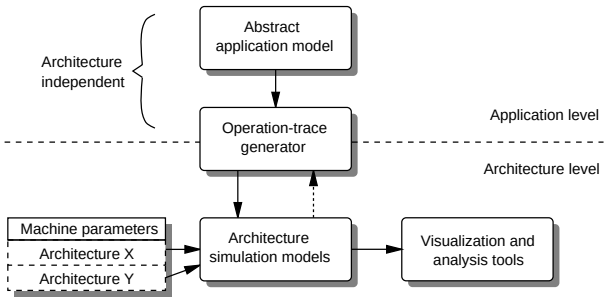


Figure 1. Overview of the Mermaid environment.

1 Introduction

Performance studies of computer architectures often involves discrete–event simulation. The simulation produces a bulk of data, and postmortem analysis is done to evaluate the performance of the simulated architecture. The Mermaid project[8], which focusses on the construction of simulation models for MIMD multicomputers, showed that it would be helpful to capture runtime effects which have an important influence on the overall performance.

In Figure 1 an overview of the Mermaid environment is given. Within this environment, the capturing of specific runtime effects should be part of the visualization and analysis tools. We decided that this should have the form of a Graphical User Interface which can display values from the simulation that have an important influence on the performance. Because it is difficult to predict in advance which values are (directly or indirectly) responsible for performance issues, it was decided that the GUI should have a very flexible nature.

The architecture models of the Mermaid project have been implemented in the special purpose computer architecture simulation language Pearl[7]. Because it was our intention to have a flexible GUI–support, it was decided to add a GUI–support library to the runtime system of Pearl[5] in such a way that it is independent of the Pearl code.

Related work in this field can, for instance, be found in the simulation of the Data Diffusion Machine[4, 10]. However, the GUI–support for that simulation is static and cannot be used for other simulations. On the other hand, another example such as the Proteus environment[2] can display statistics from generic simulations, but only postmortem. The GUI–support for Pearl inherits a property from both environments: the runtime analysis of the Data Diffusion Machine simulation, and the generic nature of the Proteus environment.

The next section gives an introduction to the structure of Pearl, and specifically the statistical analysis engine of Pearl. Section 3 gives an overview of the structure of the GUI–support library. A case study which shows how the GUI–support library is used, is found in section 4, and finally, in section 5 is a summary of this paper is presented.

2 Pearl

Pearl[7, 6] is an object–oriented discrete–event simulation language with method based communication primitives. In Pearl simulations, the objects represent modules within the computer architecture model, such as a processor, a memory, or a cache.

2.1 Objects and communication

Objects in Pearl are instances of a class. The classes of the objects determine their behaviour and are described in an imperative language, which has a syntax that is similar to C.

Objects interact with each other by sending messages. A message consists of three parts: the destination, the method and a list of parameters. The method of a message is a procedure or function which is invoked by the object when the message is received.

When a message is sent, it is queued in the buffer of the receiving object. This does not mean that the message is received immediately. The receiving object must specify that it is receptive for this kind of message. Doing so will result in blocking the execution within the object until a message of the proper type is received.

The kind of communication that is described above is asynchronous communication, which means that the sending object will not block for an aknowledgement of the message. There is, however, also a set of primitives for synchronous communication. Synchronous communication is defined in terms of asynchronous communication and a blocking receive.

2.2 The virtual timer

To maintain the correct execution order, Pearl is equipped with a virtual clock. When the simulation starts, the clock is set to 0 and as the simulation progresses, it increments.

Objects can simulate work being done by suspending execution for an amount of clock ticks. Next to waiting for messages, this presents the only other way to suspend objects. When all objects are suspended the clock will advance until the next event, i.e. the first object that was waiting on the clock wakes up. If there is no event to be scheduled anymore, the simulation terminates.

Besides waiting for just the clock, or certain messages, it is also possible to wait for either of them to occur first. This way, it is possible to simulate, for instance, interupts.

2.3 Statistical analysis in Pearl

The statistical analysis in Pearl based on a notion of the state of objects. In Table 1 an overview is given of how the state is defined.

With this notion of state, there are six different types of statistical analysis provided by Pearl: utilization, contention, profiling, call graph (critical path) analysis, average bandwidth and statistics from the Pearl-level.

waiting for clock	waiting for message	state
–	–	*ready* (if not scheduled)
–	–	*running* (if scheduled)
X	–	*wait–for–clock*
–	X	*wait–for–message*
X	X	*wait–for–clock–or–message*

Table 1. The state of objects.

2.3.1 Utilization analysis

Whenever an object is waiting for a message, it is said to be idle, and when it is waiting for the clock, it is busy. When it is in the state *wait–for–clock–or–message*, it is busy or idle depending on how it is woken up: if it is woken up by a message it was idle and if it is woken up by the clock it was busy. The total amount of virtual time spent as idle or busy is translated into a percentage, which is displayed at the end of the simulation for every object.

2.3.2 Contention analysis

The contention analysis and the utilization analysis are combined in the output, as the amount of time the objects were idle and busy is differentiated for the number of messages in the message queue. This gives an indication of how much time is spent with queued messages. If there are messages queued for a considerable amount of the time in which an object is busy, this may be an indication that there exists a bottle–neck in the architecture.

2.3.3 Profiling analysis

The virtual time spent in each method is counted so the computer architect can determine where time is spent inside the object. If it turns out that time is spent mostly in just one or a few methods, the computer architect can then decide to optimize those methods.

2.3.4 Call graph analysis

The most complicated analysis supported by the Pearl kernel is the call graph analysis. At the end of every idle period, the kernel traces back to see which objects were responsible for the objects being idle. For instance, the call–graph analysis can determine that a processor–object is actually waiting for a memory–object, even though it made a request to a cache–object (which resulted in a miss).

2.3.5 Average bandwidth

The size of the parameters at each method call is added to a total. For each two objects, the total size is accumelated

and at the end of the simulation, it is divided through the virtual time. This results in an average bandwidth of communication between objects. This can help the computer architect to decide how to place the architecture components on a board, so the communication between them does not pose a problem. Note that only the average bandwidth is used and that peak bandwidth is not taken into account.

2.3.6 Pearl–level statistics

In case there are statistics from the Pearl simulation that are not supported by the standard statistical analysis, the Pearl program must accumulate the statistics itself. A typical example of this is, for instance, the hit rate of a cache. At the end of the simulation, a special method is called in every object, in which the Pearl program is given the chance to print these statistics.

2.4 Interpreting the statistics

When running a Pearl simulation with a large number of objects, the postmortem output would be a couple of thousands of lines. It can therefore get very tedious for the designer to get a global overview of the performance issues of the simulation. A set of tools called RAPID[9] is available to gather data from vast output files, but it is still cumbersome to get a good indication of specific performance issues.

Besides it being awkward to interpret the simulation, the statistical analysis provided by Pearl does not always give any insight into *why* the performance of the simulated architecture is what it is. Often exceptional events in the simulation can result in unpredicted behaviour, and Pearl does not help the designer to trace those events. Consequently, the designer will add a print statement at every event, redirects the output to a file which can easily grow to over a few tens of megabytes. This indicates that a better way of presenting the statistics with, for instance, a graphical user interface is desired.

3 The GUI–support library

As described in the introduction, it was very important that the GUI–support is flexible in use. This was accomplished by shifting the description of the GUI to a seperate file, and thus making the Pearl program independent of the GUI. Besides that, the user should be able to make a selection of which statistics on what events or values, should be displayed. From these two requirements the framework of the GUI–support library evolved.

This framework consists of a so called canvas, the main window, with a few controls which make it possible to run the simulation in three different modes: continuously, per time step, and per scheduled event.

On the canvas, small windows can be placed, each window depicting an object. Every object can have only one window, but if the object is not interesting, it does not need to have one. Visualizing widgets, or in short *visuals*, can be placed inside the object–windows. Visuals come in various types and they display statistics from the objects.

Defining visuals is done in three steps:

1. Determining which statistics from the simulation are interesting.
2. Doing (optional) transformations on these statistics.
3. Determining in what way to present the (possibly transformed) values.

3.1 Statistics from the simulation

The statistics from the simulation that can be retrieved by the GUI–support lead to types of analysis that are quite similar to the original types of analysis without the GUI–support. The kinds of analysis that are supported are utilization, contention, profiling and analysis at the Pearl–level.

3.1.1 Utilization analysis

To determine the utilization of an object the interesting states are *wait–for-clock*, *wait–for-message* and *wait–for–clock–or–message*.

The utilization analysis by the GUI–support is slightly different from the original statistical analysis of Pearl. Instead of an object being busy or idle, there is a new definition of the state. In this new definition there are three states, called *work*, *wait* and *idle*. Their definition is related to Pearl's definition of state. An object is in state *work* when the Pearl state is either *wait–for–clock* or *wait–for–clock–or–message*. When the Pearl state is *wait–for–message*, the object is said to be *idle*. The only exception to this, is when an object is waiting for the reply message from a synchronous communication. Then the object is in state *wait*.

The problem that remains is the dubious interpretation of the Pearl state *wait–for-clock–or–message*, as it can both be interpreted as the object being *idle* or *busy*. This problem also exists in the statistical analysis of Pearl without the GUI–support. However, it turns out that the way this state is used, it is almost always interpreted as *busy*. Therefore *wait–for-clock–or–message* maps to the state *busy*.

The advantage of identifying the state *wait* is that it gives an opportunity to find objects responsible for other objects being idle. This is a property that is usualy detected by analyzing the statistics from the call graph analysis from Pearl. Making the statistics of the standard call graph analysis accessable for the GUI–support would not be difficult, but using it in a GUI would be quite cumbersome. For every pair

of objects in the simulation, a number is stored containing the statistics on the call graph analysis. That means that the size of the statistical data on call graph analysis is related to the square of the amount of objects. If this should be visualized, then a selection from these statistics must be made. The problem is that it is often hard to predict where the critical path really is. Using state *wait* is less sophisticated but gives a clear indication where objects wait on other objects. For example, the amount of time that a processor object is in the state *wait* is the sum of the time it is waiting for the cache and (indirectly) for the memory.

3.1.2 Contention analysis

As indicated before, the message queue length can be an indication of the amount of contention around an object. Whenever a message is sent to an object, it is queued in a buffer. If the object is unable to enter the appropriate method, the message remains queued. When the message queue length of an object is non-zero on a regular basis, it is an indication that the object suffers from contention. Therefore, the message queue length is the second statistic that can be derived from a Pearl–simulation.

3.1.3 Profiling analysis

Every time a method is invoked, a counter for that method is increased, and when it exits from the method, the counter is decreased. The values of these counters can be retrieved by the GUI–support and with these values, a profile of the object can be made. Because of the way the counter works, it is also possible to detect recursion.

3.1.4 Pearl–level analysis

The last means of collecting statistics from the simulation is from data at the Pearl–level. This is done by tracing the values assigned to global variables.

3.2 Dealing with statistics

Most of the values that are derived from the simulation have a temporal character; they only provide information on the simulation at the moment that they were derived. When going through a simulation step by step to get a better understanding of what is happening, this is desirable, but when the simulation is run continuously, a snapshot is not interesting. One would like to see statistics from the simulation describing performance behaviour over a longer period of the simulation. This leads to dividing values from the simulation in two catagories.

snapshot values are values that say something about the simulation at a particular moment in the simulation.

integrated values are values that say something about the simulation over some time interval.

The objective is to make integrated values out of snapshot values. This is accomplished by introducing a number of simple transformations on raw values. These transformations form only one of the three steps in the path from the simulation to the GUI. The complete path is:

- Retrieving raw values.
- Transformations on the raw values.
- Grouping values.

The last two of those steps are optional. To get a better understanding of how the transformations work, they are represented as some abstract structure. Because the intermediate results on this path form a handle to the values from the simulation, these abstract structures have been called *handles.*

3.2.1 Retrieving raw values

To get a better understanding of dealing with the values, another catagorization is made.

quantative values are values that are a quantity, such as the hit–rate of a cache object.

event values are values that describe an event being raised or not, such as a value describing whether or not a cache object is busy fetching a new cache block.

Typically, event values are of a boolean nature. However, the message queue length, which indicates the event of an object suffering from contention, can be any value greater than 0. The same goes for the method invocation counter, which also detects an event, but can be any value greater than 0. The state of an object is even worse, as it can have three enumerated values (busy, idle and wait).

For the state of an object, the solution is simple. Instead of having only one enumerated value that represents the state of an object, there are three status values, one for each state. The values are 1 if the object is in the appropriate state, and 0 otherwise.

Sometimes the enumerated value can be useful (for instance by associating every state with a color and displaying that in one visual). For that purpose, the enumerated value can be used translating to an integer that is either 0, 1 or 2.

For the message queue length and the method invocation counter things are not much different. Sometimes the actual number of messages in the queue is what is needed, while in other situations it is enough to know there is one or more

messages in the queue. The same goes for the method invocation counter. Therefore, it is possible to specify the values of the message queue length and the method invocation counter as booleans. The GUI–support takes care that values declared as boolean will be 0 if the actual value is 0, and otherwise it will be 1.

It is possible to pass raw values directly to the GUI–support library. However, this is only of use when the value in question is an integrated value. The logic behind this is simple. Snapshot values have a tendency to change very fast. Suppose we were to display the snapshot value of, for instance, the message queue length, in continuous mode. Because the message queue length changes rapidly, the value displayed would never be stable and it would be impossible to draw conclusions from it.

On the other hand, when going through the simulation step by step, it usually is to get a better understanding of what is actually happening in the simulation. This is done by looking at the events occuring at a certain moment and their effects on other events. The words "at a certain moment" already state that what is visualized, must be snapshot values.

3.2.2 Transformations on raw values

Most values derived from the simulation are snapshot values. If the simulation is running in continuous mode, it is required that the snapshot values are being translated to integrated values. This is done by sampling the value at regular intervals. With the acquired samples of the behaviour of the value over a longer period can be determined.

Currently, there are four handles that are used to transform snapshot values into integrated values. They are called *smooth*, *history*, *sum* and *average*.

The fastest way to convert a snapshot value into an integrated value is by using *smooth*. When the *smooth* handle is applied, the value that is being smoothed is sampled at regular intervals. The *smooth* handle has three parameters, the value that is being smoothed, the time between sampling and the weight for the currently sampled value. This weight is used when calculating the new smoothed value, which is the weighted average of the old smoothed value and current value. When the weight for the current value is relatively low, the smoothed value will approximate the average of the sampled values.

The second transformation that samples a value on regular intervals is *history*. Unlike what is done by the *smooth* handle, a history of these samples is kept. Such a history can either be used for further processing with handles, or it can be passed directly to the GUI. In that case, the entire range will be displayed in one visual.

The *history* handle also has three parameters. Just like *smooth*, the first two are the sampled value and the time between samples. The third parameter is the number of samples the *history* keeps.

The *sum* and *average* handles are strongly related to the *history* handle. In fact, the *sum* and *average* are only defined on *histories*. Whereas *history* goes from one value to a range of them, *sum* and *average* go from a range to one value. The names are self–explanatory: *sum* sums up all the values in a range, while *average* calculates the average.

The combination of *history* and *sum* or *average* presents the second way to convert snapshot values to integrated values. An often used technique is to take a (raw) value that can be either 0 or 1, make a *history* of length 100 from it, and define a *sum* handle on the resulting *history*. The result is an (integrated) value which gives the percentage of time the raw value was equal to 1. This technique works on all values that have a boolean character.

The advantage of using the combination of *history* with *sum* and *average* instead of *smooth*, is that if enough samples are used in the *history*, it is more stable than *smooth*, because *smooth* only averages over two values. The disadvantage is that it takes more memory, because all samples of the *history* must be stored.

3.2.3 Grouping values

The last step that can be taken before visualizing, is to group values together so they can be displayed together. The *group* handle is used for this. It doesn't do anything with the values, but only tells the GUI the values should be displayed together in one visual.

3.3 Visualizing widgets

In the GUI–support library, there are five types of visuals that make use of three different ways of displaying a value. These three different ways are: by a colour, by the length of a bar, or verbatim. Using the length of a bar is convenient when displaying quantitive values. On the other hand, it turns out, that event values are better interpreted if they are represented by a colour rather than the length of a bar. In case of step–by-step simulation, the event can be displayed as being raised or not in two colours. When simulating in continuous mode, the amount of time an event was raised can be displayed as a range of colours, but also as the length of a bar. The choice between those two depends on the characteristics of the value and what the designer wants to detect from them.

To get a clear picture of a simulation as a whole, using colours is the easiest to interpret. Especially utilization and contention can be seen in a glance if for instance the colour red is associated with high contention, and blue with low contention. However, values that are instable, would change colour all the time and reduce the clarity of the GUI drasti-

cally. Such values are only of use in step–by–step simulation or after they are first made integrated values.

3.3.1 Visualizing a value by the length of a bar

There are two visuals that use the length of a bar to display a value. They are called *graph* and *diagram*, and are displayed in Figure 2.

Graph Diagram

Figure 2. The visual–types *graph* and *diagram*.

The *diagram* takes a group of values and draws a number of bars of which the lengths are relative to the values. As the value changes, the length of the bar changes as well. The *graph* shows the recent past of a value. It scrolls from right to left and new values are being drawn on the right. The metrics of the value are omitted because it reduces the clarity of the GUI.

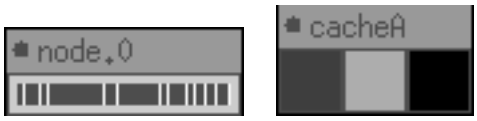
The *diagram* is exceptionally convenient if there are more diagrams displaying the same value but from different objects, so it is possible to compare. Objects that show different behaviour than the others, which are generally the interesting ones, will be noticed very easily. This is a property that all visuals that abstract from the exact value posses.

The *graph* visualizes the behaviour of a value over the most recent past. It allows the designer to look at a time slice of the simulation and monitor the statistics of which the designer has a suspicion they are influenced by special events. An example could be when the hit rate of a cache over some time period is drawn in a graph. When one of the simulated processors does a context switch, this will be clearly visible, because the hit rate will initially drop. Hence, the graph allows the designer to see these events happen at runtime rather than seeing the overall hit rate be emitted at the end of the simulation.

3.3.2 Visualizing a value by a colour

There are also two visuals that use the colour to display a value. They are called *history* and *event*, and are displayed in Figure 3.

The history does with event–like values, what the graph does with quantities. The same relation also holds for the visual–types *event* and *diagram*. When simulating step–by–step, the *event* visual is used to display events as being raised or not, and the *history* visual to display their recent past. However, the history and event visuals can do more. When simulating in continuous mode, it is also interesting to use them to display quantative values, when wanting to compare the values with similar values from other objects. Displaying a quantative value as a colour is done by assigning a colour to the upper and lower bound of the value and have a number of shades between the two extremes.



History Event

Figure 3. The visual–types *history* and *event*.

3.3.3 Displaying the exact value

There is one visual type, that displays the exact value. It is called *value*, and an example of it is displayed in Figure 4.

Figure 4. The visual–type *value*.

This visual–type is used when the exact value is required. It displays a name associated with the value (such as the name of the global variable, or a method), followed by the value. Using this visual a lot will result in a decrease of the clarity of the GUI. It should therefore be used with some care.

3.4 Agile

Describing the gui, using the structures which were described in the previous sections, is done with a list of statements in a special file. For this purpose, a language, called Agile (A Graphical user Interface LanguagE), has been developed.

As the name suggests, Agile is a flexible and powerful language. It is possible to specify a complete GUI with a minimal amount of effort. Descriptions of GUIs in Agile consists of three parts: a series of statements setting up the

environment, the placing of the objects, and the declaration of the visuals.

```
width 44;
height 44;
border 0;
color bg hotpink;
```

(a)

```
place router.* (3,3) (8,8);
wrap router.* 4;
xoffset router.* 10;
yoffset router.* 10;
```

(b)

Figure 5. Some examples of Agile statements

The first part describes the environmental parameters of the canvas. Figure 5a shows a few examples of statements that are part of this section. Things like the size of the canvas, its background colour and whether or not object windows should have a border are defined there. The second part, placing the objects, defines where the object windows will be located on the canvas. Figure 5b shows how a range of windows associated with a range of router objects are placed in a two–dimensional array with only four statements. In this particular example, windows are placed from left to right and wrap around after 4 objects. After it wraps around, the next window is placed beneath the first one again.

The third part is the creation and placing of the visuals. In Agile, it is possible to describe the raw value, the transformations, the visual type and the parameters that are required in just one statement. For instance, the line

```
display
   value sum (
      history router.*:method route (100,10,100)
   )
   (0,2) (8,2);
```

will create a visual of type *value* in each of the windows all of objects `router`. The top left corner is at the coordinates (0,2), it is 8 units wide, and 2 units high. The history contains 100 elements which are sampled at intervals of 10 ticks. So in the visual, the percentage of time that the object was in the method `route` over the last 1000 clockticks is displayed.

It turns out that besides being flexible, Agile is also very compact. Typically, files used to describe the GUI are somewhere between the 50 and 80 lines.

4 Using the GUI–support; a case study

During the design and implementation of the GUI–support library, a simulation of a multi dimensional network functioned as one of the testcases. The simulated network consists of 49 nodes in a 7x7 matrix, which are connected as either a mesh or a torus. It is possible to select between two routing strategies[1], XY routing or routing based on the Bresenham[3] algorithm (graphical routing). Messages are partially generated using a random distribution and a small percentage of all messages is always sent to the node in the center of the network, thus creating a hotspot.

Within the simulation, every node consists of 6 objects: a processor, a router and four channels. Because the processor does not produce any interesting statistics, it is not displayed in the GUI.

From the router objects, the average network contention overhead is displayed. The latency that a message would have when there is no contention, can be calculated. Every clock tick that it takes longer, is overhead. When the total overhead is multiplied by 100 and divided by the total of the expected latency, the result is a percentage of the overhead. This percentage, which is stored in a global variable, is displayed in a history visual, so when the simulation reaches a steady state, this can be detected by the history being just one colour.

The optimal configuration of a network under a given load, is when there is as high utilization and as little contention as possible. Because the router objects are only used to measure contention, the channel objects are used to analyze the utilization. There are two possible ways to do so. The first option would be to look at the time that is spent in the method that is used by the channel to pass the message to the adjacent router. However, this entity also accounts for the time that the channel is waiting on that router before it is ready to receive the message. This amount of time is contention overhead and should not be interpreted as time in which the channel object was busy. Therefore, using the state of the object is more appropriate. A history is made from *state work*, which is summed, and then used in an *event* visual.

The next step is to tune the visuals so the values are displayed in a way that the extremes of the values results in the extremes of the colours. Because all latencies are simulated by the channels and there is no routing latency, the channels can be fully utilized. This means that the interesting values from the channel utilization range from 0% up to 100%. The load that is placed on the network is rather heavy. After some experiments, it turns out that in worst case situations, there is an overhead of about 115%. Therefore, it is displayed in a range from 0 to 115%. To give as much contrast as possible, all the colours of the spectrum are used to depict the values. It only takes 5 statements of Agile to produce these visuals, of which a single one look like this:

```
display
   event(sum(
      history north.#:state 0 (400,25,126)
      [100]))
   (0,0) (2,2)
   ( blue-cyan[25]-green[25]-yellow[25]
      -orange[25]-red[25]);
```

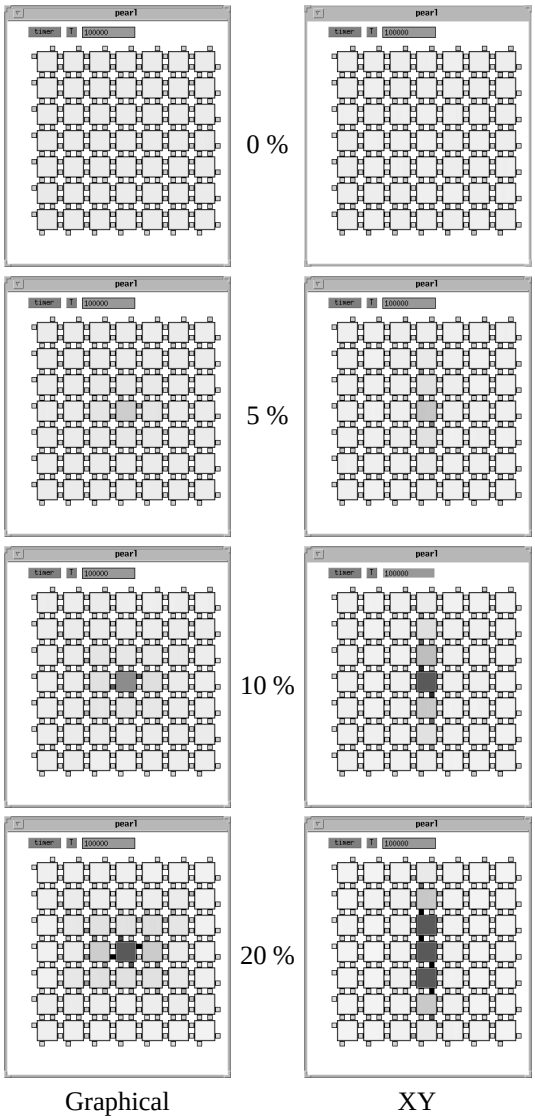


Figure 6. XY and Graphical routing in a torus

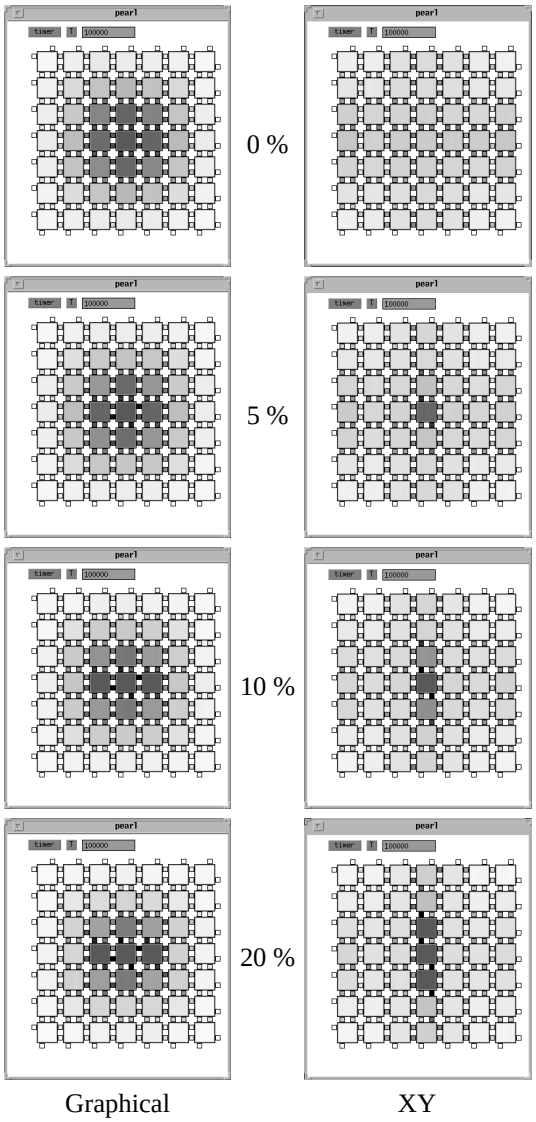


Figure 7. XY and Graphical routing in a mesh

The syntax of this statement may look somewhat frightning. It may seem that it is very error prone to specify a GUI with Agile. That is one of the reasons we intend to build a modelling environment which generates these statements.

Once the GUI has been added and the simulation is run, the performance difference between graphical routing and XY routing can easilly be visualized, especially with a relatively high percentage of the hotspot. Figure 6 compares XY and graphical routing in a torus for hotspots of 0%, 5%, 10% and 20% of all the message traffic.

With no hotspot at all, there is no difference. All routers and channels behave identically. When the hotspot is 5%, the difference is already visible. The router and the channels of the hotspot are slightly less dark with graphical routing than in the case of XY routing. Besides, the nodes north and south from the hotspot suffer from a higher contention than the other nodes. At 10% hotspot, the simulation starts developing a hot region when using XY routing, whereas the graphical routing still only gives heavy contention at the hotspot. At 20% hotspot, the difference even gets worse. The hot region of XY routing shows three nodes where the contention is very high and two where it is above average. In graphical routing it is still only the one hotspot, and adjacent nodes being just slightly darker.

However, it turns out that when configuring the network as a mesh, rather than a torus, the drop in performance is quite significant.

Without a hotspot, the performance of XY routing is just

moderately less than when the network was configured as a torus. However, with graphical routing there is a hot region of nine nodes and twelve other nodes have a significantly higher contention than in the torus configuration. When a hotspot is introduced in the center, the contention gets less. The reason for this is that the central position of the hotspot makes the total amount of hops decrease, while the number of messages stays the same. In the torus configuration this did not influence the total number of hops, because the torus wraps around and has no notion of "central". The behaviour of XY routing with an increasing hotspot is entirely different than with graphical routing because XY routing gives a better utilization of the channels at the edges of the mesh.

Although a more detailed study would be required to make a more precise statement, it is not possible to say which of the two routing strategy is better, since it depends on the circumstances.

Of course, this conclusion could also have been taken by using simulation without a runtime GUI. However, using a GUI makes it much easier to interpret the results of the simulation. Just a glance at the GUI after it reached steady state indicates around which nodes the contention is concentrated, and how heavy it is. If the designer wants to evaluate the contention of every node without a GUI, it requires digging through the output of the simulation and extract the percentages of the latency overhead. These percentages come in plain text, and although they are more accurate than interpreting a colour, it takes much longer before the same understanding is reached as with a GUI.

5 Summary

In this paper, the support for a Graphical User Interface(GUI) for the computer architecture simulation language Pearl was introduced, which allows visualization of runtime behaviour of computer architecture simulations. Because Pearl provides great flexibility in the design of simulations, it was required that the GUI–support be generic. This was accomplished by making the definition of the GUI independent of the Pearl code, which also allows one to add a GUI to older simulations, which were built before the GUI–support existed.

Originally, the GUI–support was designed to get a better and faster understanding of the simulated architecture and its performance. It turns out, that it can be used in several situations in which knowledge about a computer architecture has to be aquired or passed on to someone else. Such a situation can, for instance, also be part of an educational process.

The GUI–support can be used in two distinct modes. In case the designer wants to get a better understanding of the simulated architecture, it is possible to go through the simulation step by step, so the events that are greatly effecting the performance of the simulated architecture can be detected as they occur. A more broader view is given when the simulation is run in continuous mode. In this mode, an important part of the GUI–support is used which allows integrating raw values from the simulation over a certain time interval.

Because Pearl is an object oriented simulation language, the GUI is based on the objects that exist in a simulation. The designer specifies the GUI in a compact language called Agile, which creates the building blocks of the GUI, such as transformations on values and the widgets to visualize the statistics, called visuals. Visuals come in various types and every type has the ability to visualize specific characteristics of the values from the simulation, such as the behaviour in time, or just a snapshot of a value.

Although there is still a lot of work that can be done on the GUI–support, the case study that was discussed in this paper showed that clear runtime visualization of computer architecture simulations can be accomplished with simple building blocks. Combined with the underlying flexibility of Pearl, this makes the GUI–support a powerful tool and a great asset to Pearl.

References

[1] Didier Badouel, Charles A. Wüthrich, and Eugene L. Fiume. Routing strategies and message contention on low–dimensional interconnection networks. Technical report, Computer System Research Institute of the University of Toronto, 1991.

[2] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William A. Weihl. Proteus: A High–Performance Parallel–Architecture Simulator. Technical report, Massachusetts Institute of Technology, September 1991.

[3] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[4] Erik Hagerstein, Anders Landin, and Seif Haridi. DDM – A Cache–only Memory Architecture. Reseach report R91:19, Swedish Institute of Computer Science, November 1991.

[5] Henk C. Kok. Visualizing Computer Architecture Simulations — Graphical User Interface Support for Pearl. Master's thesis, University of Amsterdam, 1996.

[6] Henk Muler. *Simulating Computer Architectures*. PhD thesis, University of Amsterdam, 1993.

[7] Henk Muller. Pearl: A Language for Architecture Simulation, February 1993.

[8] Andy D. Pimentel and L. O. Hertzberger. An architecture workbench for multicomputers. In *Proc. of the 11th International Parallel Processing Symposium*, pages 94–99. IEEE Computer Society Press, April 1997.

[9] Andy D. Pimentel and L.O Hertzberger. Rapid: Rapid interpretation of data. Technical report, University of Amsterdam, Januari 1997. CS–97–01.

[10] Paul W.A. Stallard, Henk L. Muller, and David H.D. Warren. Performance Evaluation of Parallel Programs on the Data Diffusion Machine. *Proceedings of Performance Evaluation on Parallel Systems PEPS'93*, pages 94–101, 1993.