ColSpace VERSION 0.3 Tutorial

9/13/2009

Table of Contents

1. Introduction 2
2. How to run ColSpace 3
3. Basic operations – a quick example 4
4. Right-click (popup) menu 10
5. Ports and PortCells 13
6. Loops/iterations 14
7. Delay issues 17
8. Handling project files 21
9. Libraries 23
10. Fidelity-compromising transformations and alternative designs 24
11. Naming issues 26
12. Performance 28
13. Simulation and analysis tools 29
14. Known bugs 30
15. Features not yet available (but could be expected) 31
16. Case Study 32
17. c2colspace – creating ColSpace project from C source code 46

1. Introduction

Many applications in various disciplines are highly computationally-intensive. A direct implementation is either not possible (being too costly) or cannot meet desired requirements (power/performance). In such circumstances, the algorithm must be changed to meet requirements while the impact on application fidelity is kept to a minimum. However, both algorithm and hardware designers lack sufficient domain knowledge to reason about various tradeoffs.

ColSpace is a tool that enables both algorithm metrics (i.e. application fidelity) and implementation metrics to be co-optimized during early design exploration. The enabling data structure, the hierarchical dependency graph (HDG), is able to represent both the algorithm and the implementation architecture, so algorithm designers and hardware designers can explore the collaborative space (ColSpace) together, trading off various metrics while finding the best overall design.

The implementation metric currently modeled by ColSpace is latency (future versions will include die area costs, and power).

Once you have downloaded the zip package, let's take a look at its structure.

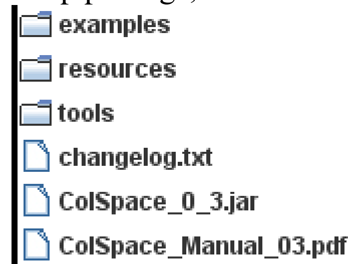


Figure 1. Directory structure of the downloaded zip file

ColSpace_0_3.jar is the binary to launch ColSpace. That pdf file is just this tutorial you are reading. *Changelog.txt* lists the difference from the previous 0.2 release.
Some example projects are provided in the /example directory. Three projects (bubble sort, GVF and SRAD) are under the /examples directory. Also included is a library "comm._lib" which can be loaded into the Library view using the "loadProjLib" button. The directory /resources holds some images for display purposes. In the /tools directory you will find a very useful C to ColSpace converter, which allows you to automatically create ColSpace project from C source code. For details, see Chapter 18.

2. How to run ColSpace

Extract the zip file into any directory you want. Then use command line tool to enter that directory, and type the following command:

```
java -jar colspace_0_3.jar
```

The GUI should appear without any problem. It looks somewhat similar to this:

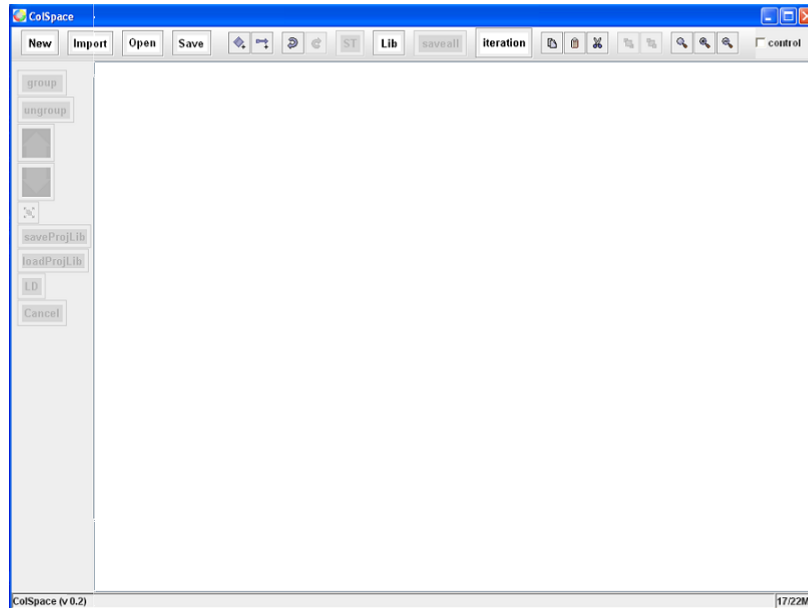


Figure 2: ColSpace default user interface

In some systems, you can skip the step of typing command line. Instead, you simply double-click the jar file. However, this approach seems to work only on versions prior to JDK 1.6. Command line is recommended.

3. Basic operations – a quick example

Click on "insert" button on top toolbar to insert a new empty node.

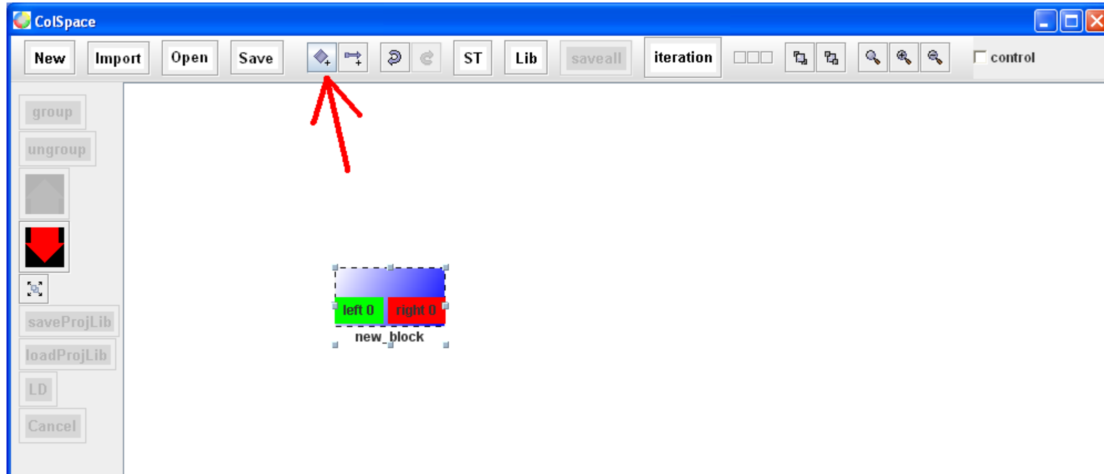


Figure 3: Inserting a new node

Insert another node by repeating the process. You can move a node around by clicking on it, holding down mouse and dragging it around.

Figure 4: Moving nodes

Use your mouse to draw a rectangle surrounding the two nodes we have just created. Both of them will be selected (group select).

Figure 5: Group selecting

Right click on a node to bring up a context menu, from which you can choose what operations to perform on that node. Click anywhere else to make it disappear.

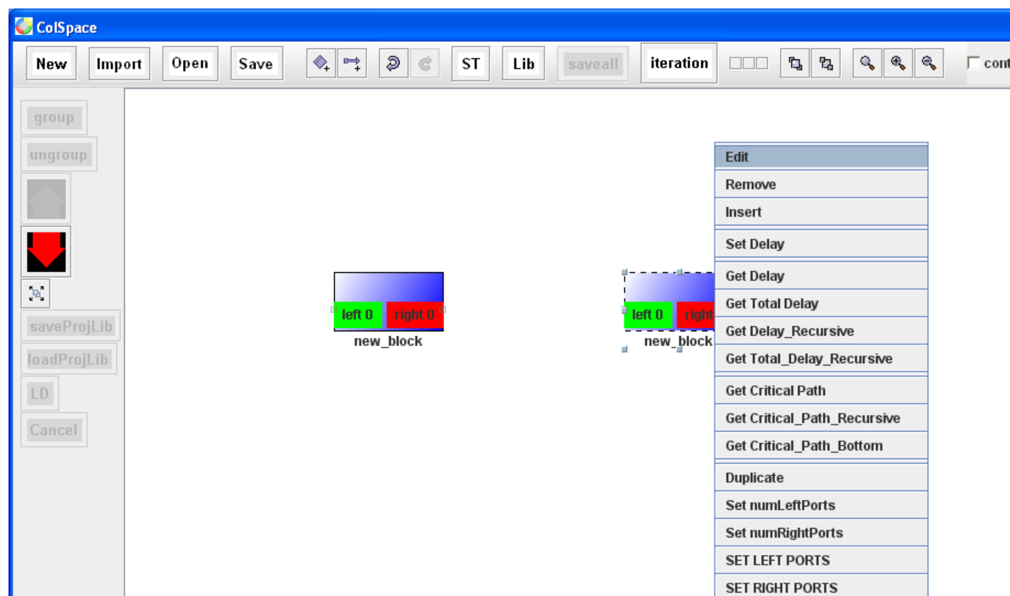


Figure 6: Context menu

A **port** is a small square aligned on the left or right sides of a node. Initially they are named like “left n” or “right n”, where n is the index. You can manually change the name of a port through “SET LEFT/RIGHT PORTS” options in right click context menu.

Now click on port “right 0” of the node on the left, hold your mouse button down and drag an arrow to the port “left 0” of the node on the right. Release mouse and an arc is created. A solid line indicates a data dependency (target depends on data from source),

while a dashed line indicates a control dependency (target conditionally depends on the results from source, which is normally a branching operation). For graphs that involve only data dependencies, data will be generated from inside the source node and delivered into the target node. The same structure is employed inside every node to deliver data to or receive data from a port. Users can toggle the dependency type of an arc by right clicking on an arc and choosing the desired type.

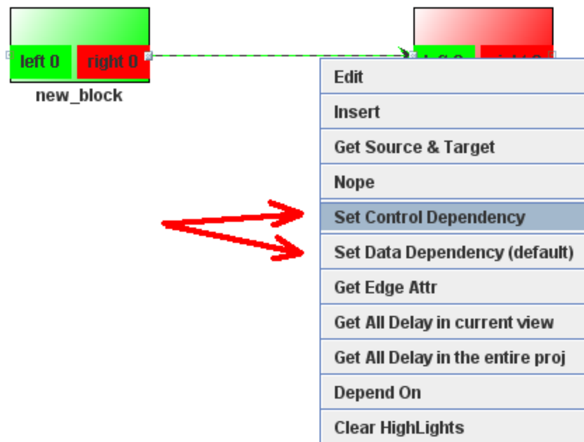


Figure 7: Set control/data dependency

Click on the edge and hit "delete" on your keyboard will remove that edge.

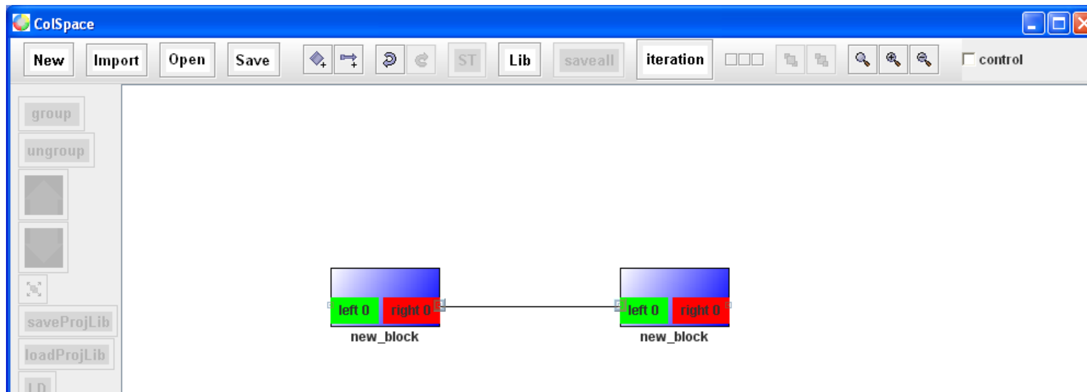


Figure 8: Connecting nodes with edges

Now group select both nodes. Notice that the "group" button in the left toolbar is enabled. Click it.

Figure 9: Grouping

Now you have created your first group node. ColSpace will prompt you to enter a new name for the group node, since a name is an indication of its structure. This group node has to be named differently from "new_block" because it now has a different structure. In ColSpace, **name is an identifier of a node's implementation**. If you need several instances of the same structure, they will have the same name but different internal IDs.

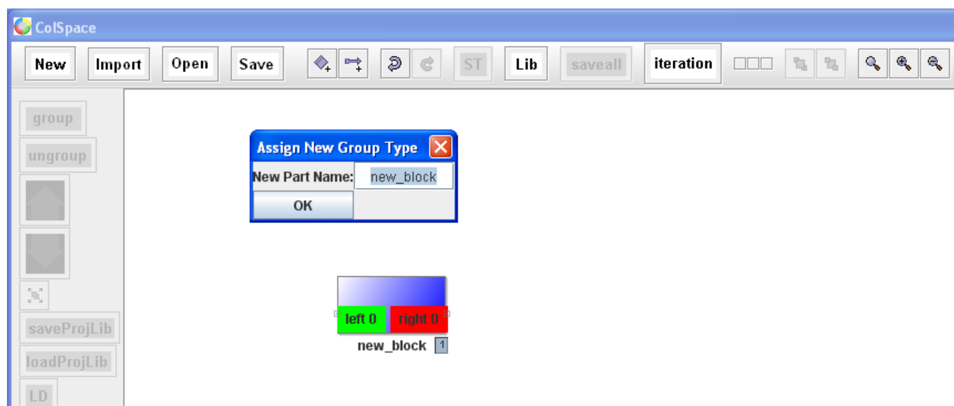


Figure 10: Renaming nodes

When you move your mouse to the corner of a node, your mouse indicator becomes a resize-arrow, allowing you to do resizing.

Figure 11: Resizing

The "navigate down" button is enabled whenever you select a group node. Click on it to navigate down one level in the hierarchy.

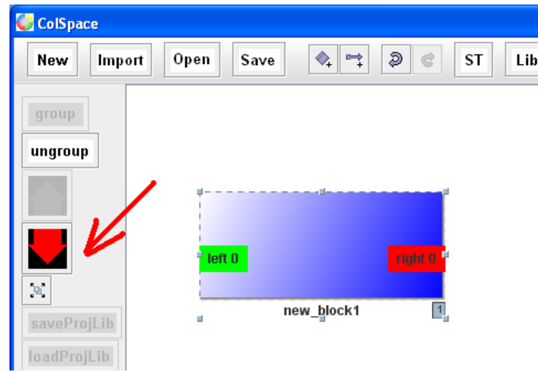


Figure 12: Navigate down

You will see some nodes named “left x” or “right x”, depending on the number of ports of the group structure. They are **Port Cell**s (see Section 4). Basically they constitute the interface through which to communicate data with outside, so that we know exactly how an edge is connected on both ends across a node boundary. Ignore them for now.
The “navigate up” button is enabled whenever you are inside a group node. Click on it to navigate up one level in the hierarchy.

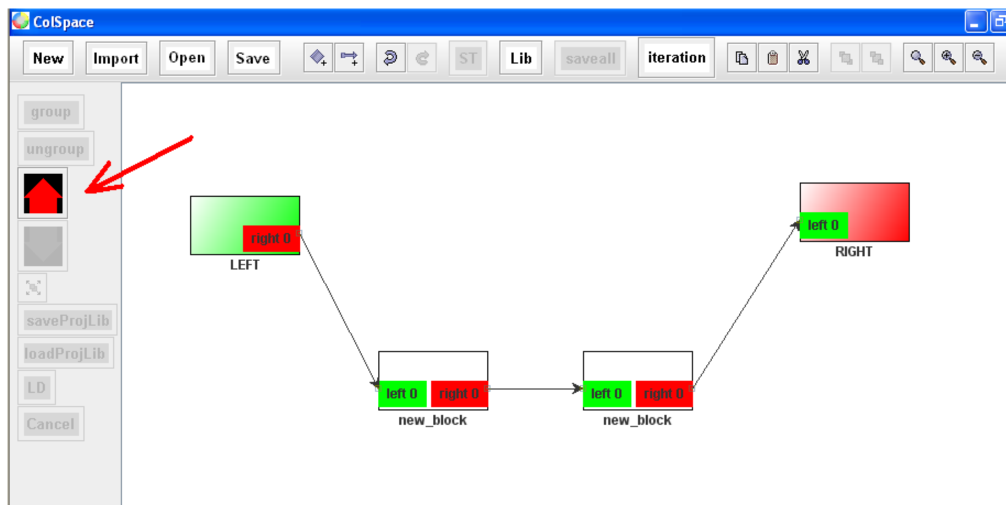


Figure 13: Navigate up

“Ungroup” will dismiss the group structure and take you back to the status before the grouping operation. Note that port cells will not be displayed because there will be no more “ports” after ungrouping. Port cells are not actually nodes, but a virtual connector.

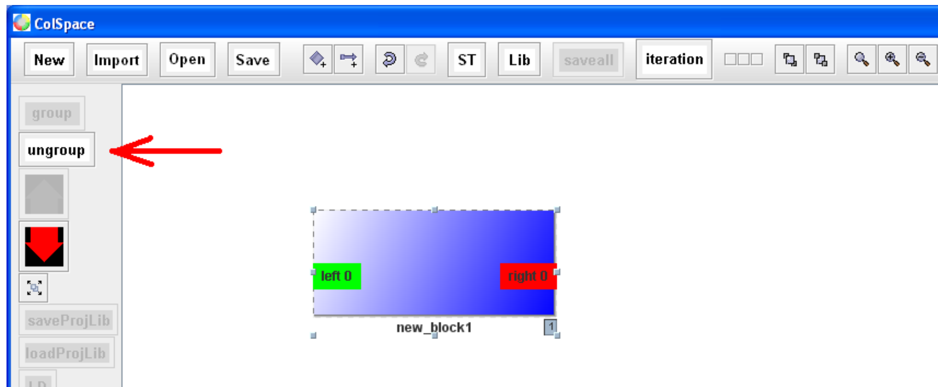


Figure 14: Ungrouping

Basic operations:

Left click to select node

Double click to rename node

Group select by dragging a rectangle around nodes

Right click to bring up context menu (different menu depending on what you click on is node or background)

Move a node by dragging it around

Connect nodes by dragging an edge from source port (i.e. output port of source node) to target port (i.e. input port of destination node).

Remove an object by selecting first, and then hit DELETE on your keyboard.

For an overview of the functions of toolbar buttons, please refer to Fig. 15.

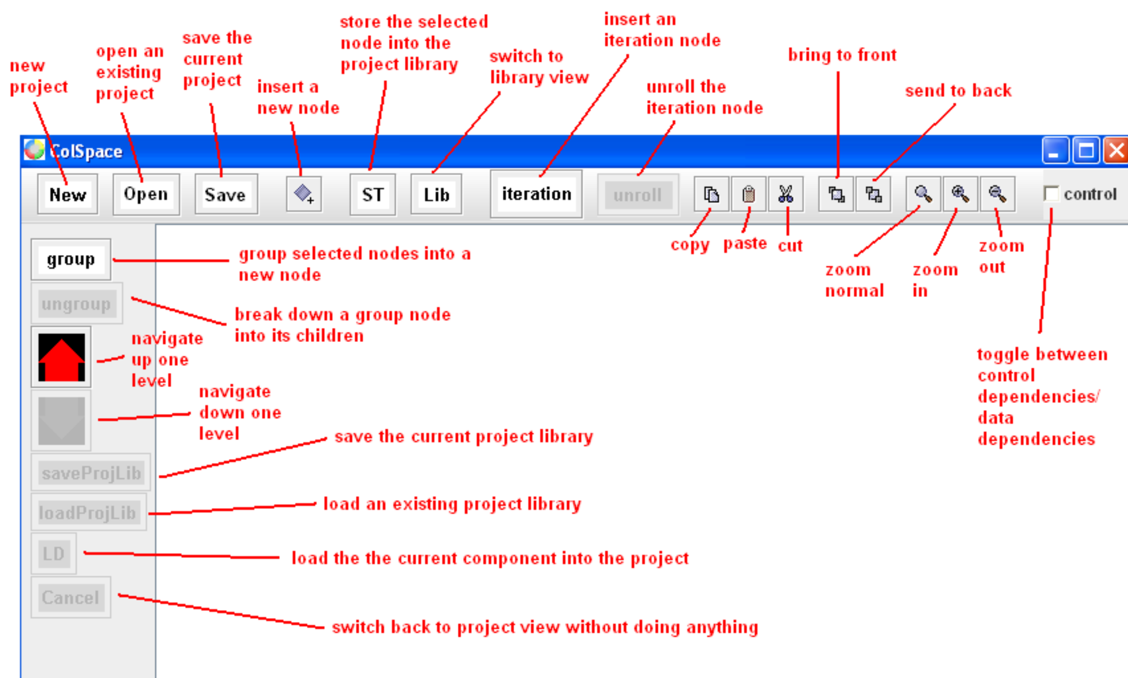


Figure 15: Toolbars

4. Right-click (popup) menu

There are three types for popup menu, depending on what you right-click on.

1) If you right-click on the background:

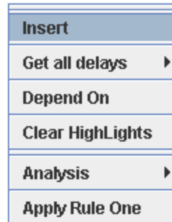


Figure 16: Background menu

Depend on is a useful tool to quickly find the dependency between two arbitrary nodes. Simply supply with the IDs of the first node and second node, and you will see whether the first on is dependant on the second. Note that the ID is not the name appeared below each node (which is its TYPE). ID is a unique identifier for each node within the entire project. The ID of a node is shown on the first row of the right click popup menu of that node (marked by two arrows).

2) If you right-click on an edge:

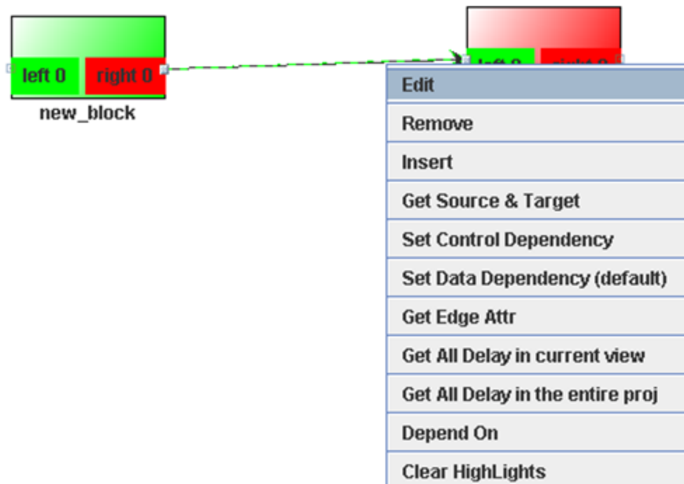


Figure 17: Edge menu

Set Control Dependency: set the currently selected edge to be control dependency (instead of data dependency).

3) If you right-click on a node:

--->cell0:cell3<---
Edit
Remove
Insert
Set Delay
Set as original design
Set as alternative design
Replace with alternative design
Replace with original design
Get Delay
Critical Path
Duplicate
Set #Ports
Set Ports
Exploit Parallelism
Set # of inputs consumed/iter
Set # of outputs produced/iter
Set as Loop
Set # of Iterations
Set as while condition
Implement on
Get Location
Get all delays
Depend On
Clear HighLights
Analysis
Apply Rule One

Figure 18: Context menu

Get Delay | Get Delay
Critical Path | Get Total Delay
Duplicate | Get True Total Delay
Set #Ports | Get Delay_Recursive
Set Ports | Get Total_Delay_Recursive

Figure 19: Get Delay submenu

Critical Path | Get Critical Path
Duplicate | Get Critical_Path_Recursive
Set #Ports | Get Critical_Path_Bottom
Set Ports | Get Critical_Path_Betw. A & B

Figure 20: Critical Path submenu

Implement on | CPU
Toggle Ordered | ASIC

Figure 21: Implement On submenu

Get all delays | Get All Delays in Current View
Depend On | Get All Delays in the Entire Proj

Figure 22: Get all delays submenu

Get Location | Simulate latency!
Get all delays | change latencies and run!
Depend On | change (globally) parameters
Clear HighLights | print all cell types
Analysis | print component contribution/sensitivity

Figure 23: Analysis submenu

Most of the menu items are self-explanatory. Those requiring further explanation are listed below.

--->cell0:cell3<---: the absolute path of the node. It means that this node (cell36) is a child of cell64, who in turn is a child of cell72. Although the cell ID can unique determine any cell, it is more efficient to use such a path to search for a certain node and highlight its topological position in the hierarchy.
Edit: rename
Insert: Insert a new node right where you click
All Get Delay, Critical Path and Exploit parallelism are discussed in *Section 6. Delay Issues*.
Duplicate: deprecated.

Set # of inputs/outputs consumed/produced per iteration: are specific to synchronous dataflow graph for proper scheduling of components. It can also be used in constructing a pipeline. (Currently not in effect).
Set #Ports allow users to change the number of left and right ports of a certain node.
Set Ports are used to change the names of ports of a certain node.
Get Location prints the location of current node (expressed in upper left corner coordinates) as well as its width and height.
Get All Delays in Current View / Get All Delays in the Entire Proj list the delay values of all the nodes displayed in current view or in the entire project.
Depend on same as in right-click popup menu on the background.
Clear Highlights is used in combination with some other functions. For example, **Get Critical Path** highlights the critical path found. **Clear Highlights** can be used to clear the highlights created by certain functions. Note that highlights will be automatically cleared when you left that hierarchical level (either up or down).

5. Ports and PortCells

To unambiguously represent the dependencies across hierarchical levels, we have installed input/output ports on each node for anchoring dependency arcs. Furthermore, the port interface provides extensibility because nothing needs to be done outside the component as long as we keep its port interface stable, no matter how the underlying structure changes. A change of port interface is also easy by writing a port adapter.

A **PortCell** is a node representing a port of another node. For example, a node has two left ports and three right ports. It looks something like this:

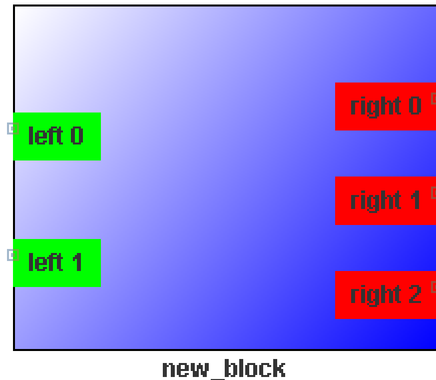


Figure 24: Ports of a node

When we go inside this block, we'll see the following:

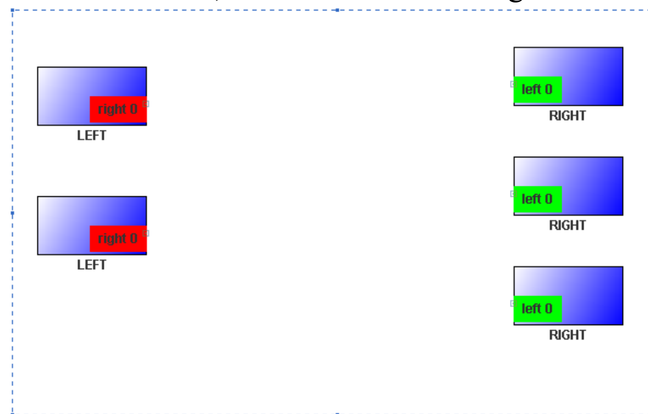


Figure 25: Portcells

These five nodes are called "PortCell"s. They represent the five ports in the previous view. They are here for connecting internal structures (we could add some nodes inside new_block and connect them to the portcells). In this way we guarantee the dependencies across hierarchical levels are preserved and accurate.

PortCells are primitive nodes which do not contain substructures. If you descend into a PortCell, you will see nothing. Unlike regular nodes, PortCells do not contain further PortCells. However, PortCells do have ports. A right PortCell has a left port, and a left PortCell has a right port.

6. Loops/iterations

To create a loop node, click on “iteration” button. You will be asked to enter the number of iterations for that loop node.

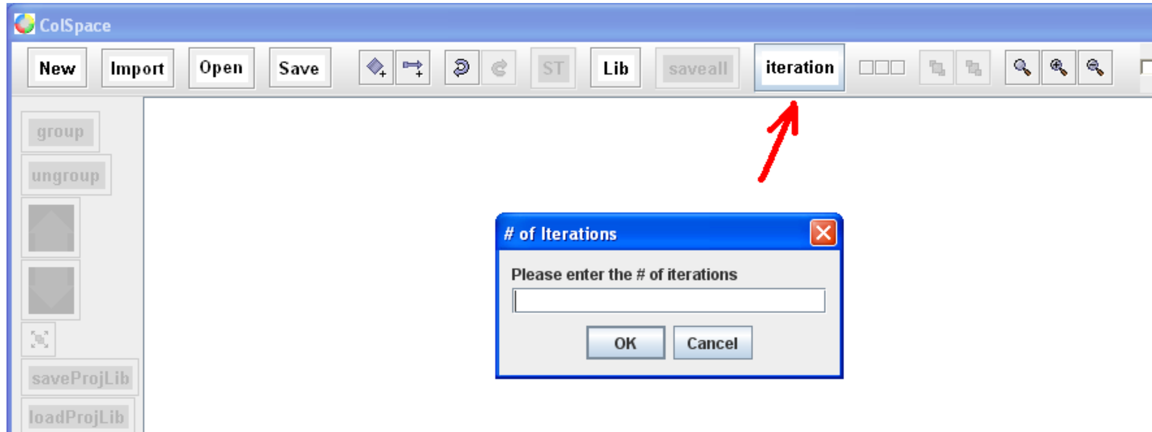


Figure 26: Iteration

The number of iterations is always a positive integer (1 for non-loops). However, the user can enter 0 if he/she wishes to enter that number later during latency evaluation, possibly because the number is yet to be determined. ColSpace will remember this value as an undefined parameter. The user can proceed with other operations but will be prompted for the value when it is needed by the tool, e.g. when calculating latencies that demand the latency of this node.

The number of iterations will appear at the upper left corner of the node.

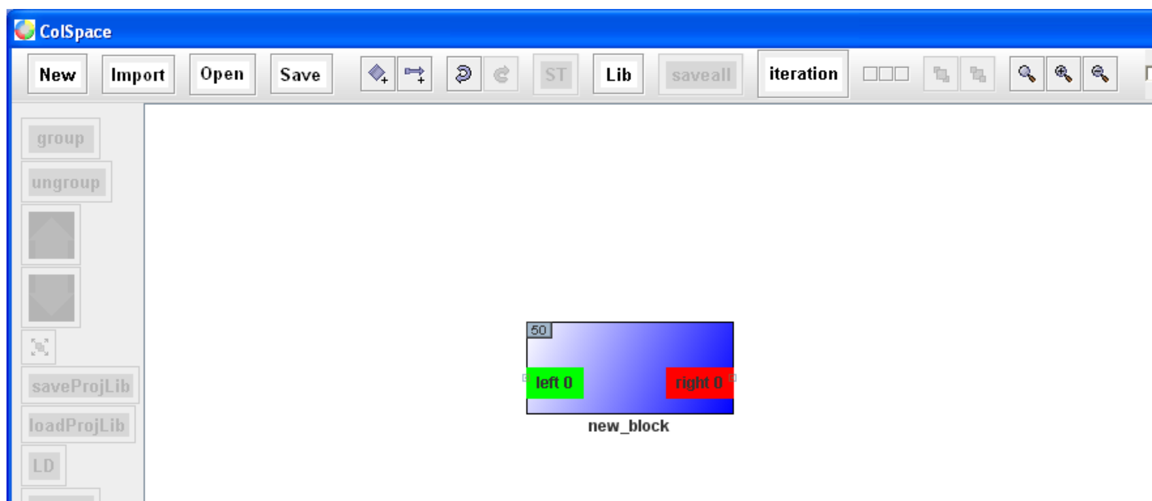


Figure 27: The number of iterations at the upper left corner

Right click on any node, you will find “Set # of Iterations” for you to conveniently change the number of iterations for that node.

You will see an “unroll” button if you are inside an iteration node. It allows you to adjust the granularity of an iteration node, as it may not be necessary for an iteration to complete before beginning the next. For example, Figure 28 represents one iteration of a loop to calculate the variance of a dataset with *N* samples, requiring *N* iterations. If we assume that each node has a latency of 1 time step, the iteration latency is 5 and the total loop latency is 5**N*. By unrolling this loop once (reducing the number of iterations to *N*/2) and restructuring the graph based on the real dependencies (Figure 29), we note that the iteration latency has only increased to 6 time steps, providing a new total loop latency of 3**N*.

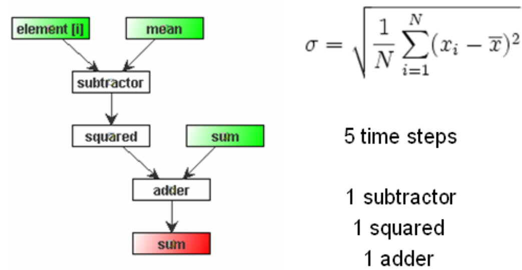


Figure 28: Variance calculation: one iteration.

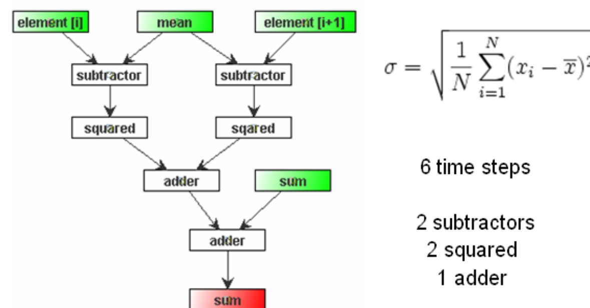


Figure 29: Variance calculation: unrolled once.

This unrolling can continue to further reduce latency (like the example above, it does not affect the application fidelity) but with diminished returns and at the cost of additional resources (i.e. die area).

In the previous 0.2 release, you could specify whether a node is *unordered*, *concurrent* or *pipelined*. These features are removed from this release because they can be completely inferred from node dependencies, input consumption and output production rates. Allowing the user to specify these properties will lead to inconsistencies in the representation. We delegate this job to node level dependencies knowing that it is a more general and powerful approach. To achieve that, we must first define dependencies across iterations:

We made a rule that if an input (left) port and output (right) port of an iteration node share the same name, there is a dependency between the output port at iteration i and input port at iteration i+1, that is, the input port takes its value from the output port in the previous iteration. From this rule we can deduct two guidelines:

1) For an *ordered* and *nonconcurrent* node, the output names should be a subset of the input names. If an output is not used as an input, its value is overwritten every

iteration but the last one. Arrays are an exception: values can be written to different locations in the array.

2) *Unordered* and *Concurrent* iterations generally should not share names among input and output ports, because that creates dependency that prevents parallelism. However, in some cases the progression of a port value can be statically determined:

```
for (i=0; i<=99; i++)
  result += data[i];
```

This i can appear at both input and output ports and still allow the iterations to be *unordered*, because all i's can be statically determined prior to runtime.

The concept of pipeline is introduced in Section 7.

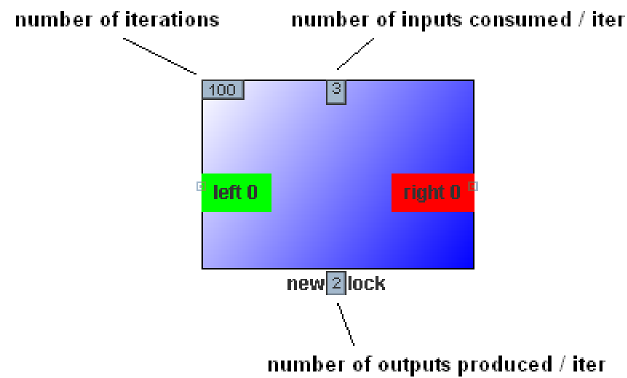
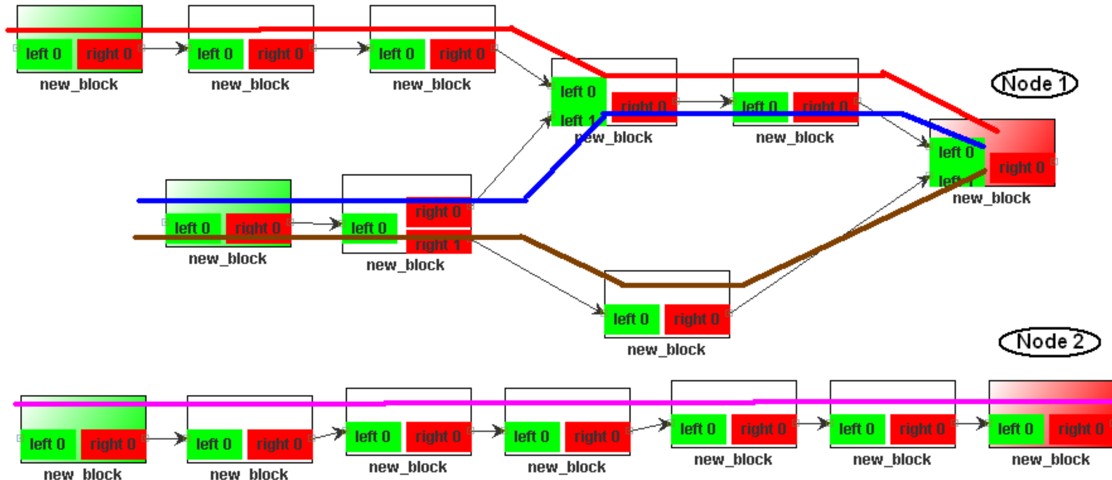


Figure 30: Corner notations of a node

7. Delay issues

Here are some delay related menu items from the popup menu.

Set Delay: set the delay of the node (doesn't work for group nodes)
Total Delay: the longest visible path (in current hierarchical level) from any input to self



Consider the above example; there are three paths from any input to output node 1 (marked respectively in red, blue and brown). Suppose all nodes have equal latency of 1, then Get Total Delay of node 1 should give the latency of the red path, which is 6. The latency of the node itself (node 1) is also included.

Delay Recursive: the longest bottom level path between any input and output after flattening the selected node.

For example, the above HDG is the underlying structure of a group node (say, node 3). So Get Delay Recursive on node 3 shall give the latency of the pink path (with latency 7, it is the longest path from any input to any output). This is only true when all new_blocks here are basic nodes with no further underlying structure. If any one of them is compound node, it has to be ungrouped till it is only composed of basic nodes. Then the longest path evaluated.

Total Delay Recursive: the longest bottom level path between an input and an output of self after flattening the entire current level view.

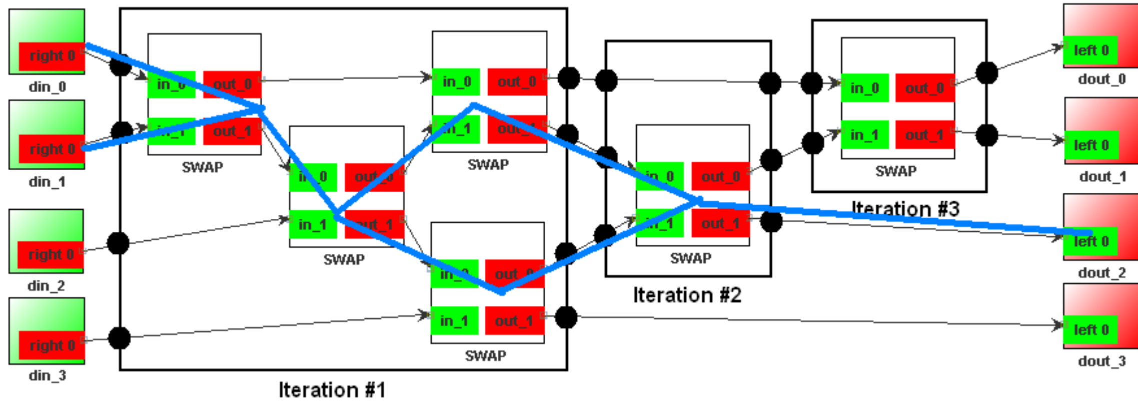
Somewhat similar to Delay Recursive, but with a global perspective. For example, assume that node 1 is a group node, then Total Delay Recursive on node 1 will do a Delay Recursive on node 1 as well as all the nodes preceding it in current view (all the nodes on red, blue and brown paths). It can be understood as the delay recursive up to and including all the output nodes of node 1.

One unresolved **bug** is that **Delay Recursive** and **Total Delay Recursive** do not work correctly on iteration nodes. To be honest, I have not figured out what it means to perform delay recursive operation on an iteration node and why it is useful.

Get Critical Path/ Get Critical Path Recursive/ Get Critical Path Bottom list the path (an ordered list of nodes) that produces the pertinent delay value, instead of giving the delay value.

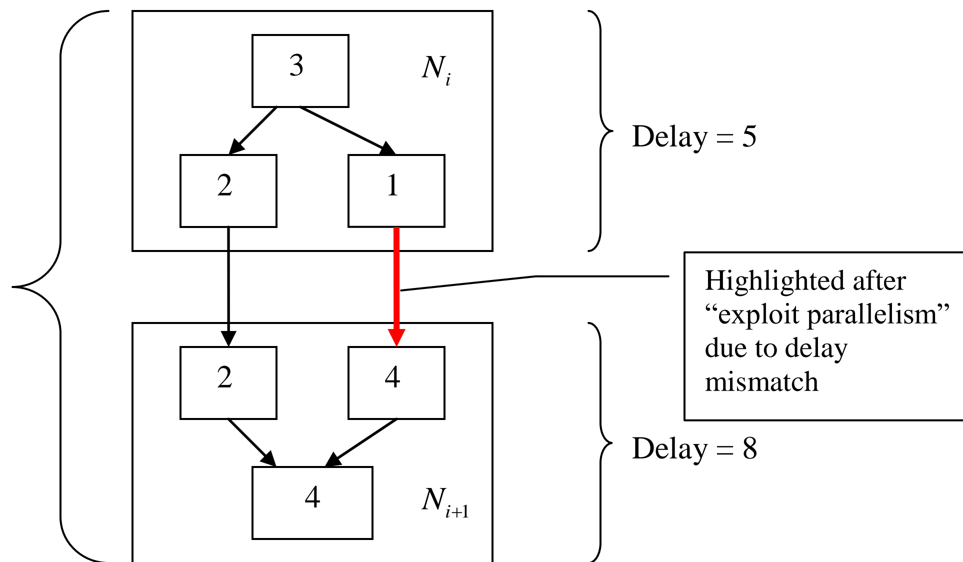
Get **Critical Path** list the path used to calculate **Total Delay**.

Both **Get Critical Path Recursive** and **Get Critical Path Bottom** use the path that calculates **Total Delay Recursive**. The key difference is that **Get Critical Path Recursive** list the current level representation of that path, while **Get Critical Path Bottom** list that path without any modification. Here's an example:



For dout_2, there are four **Total Delay Recursive** paths here illuminated in blue. **Get Critical Path Bottom** will list all four of them, while **Get Critical Path Recursive** will only list din_0->Iteration #1->Iteration #2->dout_2 and din_1->Iteration #1->Iteration #2->dout_2. These two paths represent the four bottom paths in the current level. Note that all critical paths of equal lengths are listed.

Exploit parallelism: parallelizing an algorithm into concurrent chunks is an efficient way to improve performance. However, identifying possibilities of parallelism across hierarchical levels is difficult. It'll be much more efficient for an algorithm designer to setup the dependencies and for ColSpace to explore the design space. If there exists a possible optimization, the pertinent components will be highlighted, suggesting the algorithm designer to go down that component and to do a partition.



total delay (N_i) = 5, total delay recursive (N_i) = 5

total delay (N_i) - total delay recursive (N_i) = 0

$\neq$

total delay (N_{i+1}) = 13, total delay recursive (N_{i+1}) = 12

total delay (N_{i+1}) - total delay recursive (N_{i+1}) = 1

Exploit parallelism detects delay mismatch by comparing the following two quantities:

`total_delay(N_i) - total_delay_recursive(N_i)`

and

`total_delay(N_i+1) - total_delay_recursive(N_i+1)`

N_i and N_{i+1} are two immediately neighboring nodes. A hidden parallelism exists if and only if the above two values are not equal.

Here is a summary of various types of delays, paths and their definitions.

Table 1: Delays, paths and their definitions

	Delay / Path (on Node c)	Definition
1	Delay	Max path delay between any inputs and outputs (portCells) only at level just below. For atomic cell, it is hard-coded.
2	Total Delay	In its existing level, trace toward left to find all possible paths leading to self (until reach dead end, can be non-portCell), max path delay (including self delay) = Total Delay
3	True Total Delay	In its existing level, trace toward left to all input

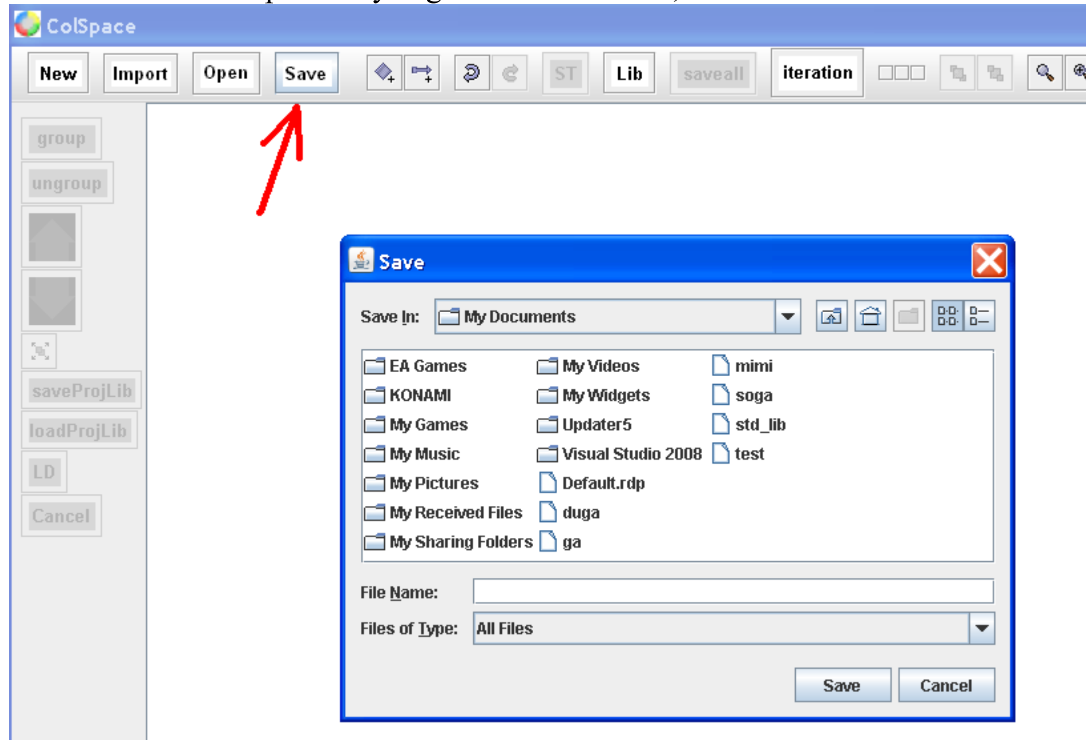
		portCells (must start with a portCell), return the max path delay (including self)
4	Delay Recursive (on c)	Starting from right portCells and trace left, only touch atomic cells, remain under the cell c, what's the max path delay.
5	Total Delay Recursive	Similar to Delay Recursive, but at one level higher, remain under c's parent.
6	Critical Path	the path which corresponds to Total Delay
7	Critical Path Recursive	the path which corresponds to Total Delay Recursive
8	Critical Path Bottom	Critical Path represented on the bottom hierarchy, made of all atomic cells

8. Handling project files

- **Creating/loading projects**

There are two ways to create a project in ColSpace.

To create a project using GUI, you simply add components, assign metrics to them and connect them with dependency edges. When finished, click on the "Save" button.



Then the project will be stored as a standard-formatted project file that ColSpace is able to understand (so that you can open it later).

Alternatively, you can write a standard-formatted project file directly without resorting to GUI. The format of the project file is as follows:

```
digraph name_of_graph {
      one or more node descriptions
      one or more edge descriptions
}
```

Node description:

```
      node_id [type="type_name", group="groupname", level="lv",
pos="x,y", iter="i", latency="lt", size="w,h", concurrency="c",
leftports="leftportsnames", rightports="rightportsnames",
leftPortCellIndex="d_l", rightPortCellIndex="d_r"];
```

Among these items, "level", "latency", "iter", "concurrency" can be omitted (default values will apply).

"leftPortCellIndex" is set to -1 if the node is not a left port cell, set to n (n>=0) means it is the #n-th left port cell (starting from #0). "rightPortCellIndex" has similar meaning. Notice that for an individual node, only one of the two values can be greater than or equal to zero, because one node can't be both left and right port cell. They can be both -1, indicating that it is not a portcell.

Groupname is in the form of "`rootparent:parent1:parent2:…:direct_parent`". For a root node, its groupname is an empty string. "level" indicates the absolute level of depth. Root level is level 0, the next level will be level 1. "iter" denotes the number of iterations of that node, if the pertinent node represents an iteration. "concurrency" also applies to an iteration node only, denoting the number of iterations that can be run concurrently.

Leftports and rightports are strings that store the name of left ports and right ports. They are arranged in the order of their index, and separated by commas.

Edge description:

```
Absolute_path_of_source_node -> absolute_path_of_target_node [(dash)];
```

`Absolute_path =` $node_id_0 : node_id_1 : ... : node_id_k$

The last node_id should refer to a portCell of the source/target node. The optional "dash" in square brackets means that dependency is control dependency. Otherwise (nothing within brackets) it is data dependency, which is the common case.

We provide the text input method in the hope that generation of dependency graph can be fully automated. For example, we can write a tool that extracts the dependency information from a C source code. However, our ultimate goal is to completely overthrow conventional programming language and to promote using ColSpace (we do not exclude the possibility of generating the project files using some scripting language), since it is more intuitive than languages like C in dealing with parallelism.

- **Saving projects**

Simply click on the "Save" button and give it a name, and you're done. It will be in the same standard format as used above.

Last open/save directory is remembered.

- **Examples**

There is an example project file "example_project" and its graphical representation "example_project.bmp". They demonstrate how you should write a correctly formatted project file.

9. Libraries

There are two types of libraries: standard and project. Standard library will automatically get loaded when you start ColSpace. There's only one standard library, and it's the same no matter what project you are working on. Standard library is meant to be read only. It is stored in a file called "std_lib" under the /resources directory. Project library, as indicated by its name, is project specific. It is empty on ColSpace startup, and does not appear in your lib view. However, you can choose to load a project library file whenever you want. In your project view, whenever you store a new component into a library (by clicking on "ST" or being prompted by ColSpace to rename a component you've just modified), that component always goes into your project library. Therefore, project libraries are writable. However, there is currently no way to remove a component from a project library (will be available later).

Click on the "Lib" button will take you to the library view. Here you can browse through all the components from standard library and project library. You can view the hierarchical structure of every component just like in project view, and the only difference is that the library view is READ-ONLY. You can choose to load a specific component into the project you are currently working on. You can also save the current project library ("saveProjLib") into a library file which you can retrieve later (by using the "loadProjLib").

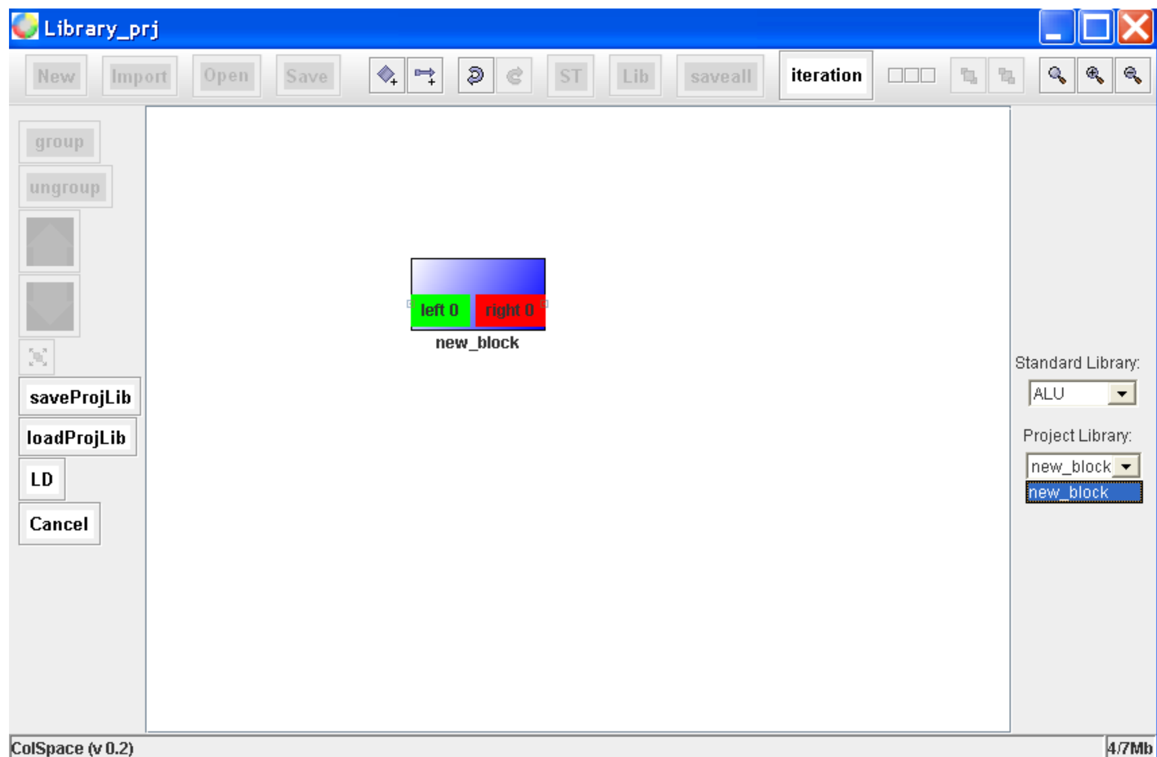


Figure 31: Library view

Format of library files is very similar to project files. However, in a library file, there might be multiple "digraph" bodies, each representing exactly one component.

10. Fidelity-compromising transformations and alternative designs

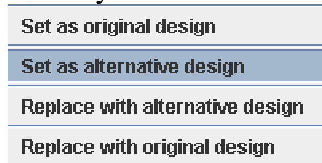
A primary new addition in 0.3 is the automation of applying fidelity-compromising transformations. The goal of such transformations is to reduce latency. To that end, computation-intensive algorithms are replaced by simpler alternatives that sacrifice accuracy. A good source of such alternatives is mathematical approximations. For instance, $\frac{1}{1+x}$ can be replaced by $1-x$ when the absolute value of x is small so that the expensive division can be avoided.

The automation process involves three stages: first, identifying the pattern within the HDG where transformations can be applied; second, applying the transformation by altering the HDG structure; third, evaluate the latency as a result of the transformation. All three stages are automated in our tool to various degrees. While currently the pattern matching stage only automates polynomial approximations, the applying and evaluation stages are fully automated to suit all types of transformations.

The procedure is as follows:

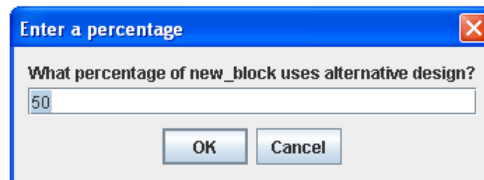
1) Mark the node you want to apply transform to as "original design" using the context menu.

2) Change the node to your desired structure after transformation. Mark it as the "alternative design" using the context menu. Currently only one alternative design is supported. Remember the ports of the original design and the alternative design should match perfectly otherwise the functionality will be affected.



3) To manually switch between original and alternative design, use the corresponding menu items.

4) To evaluate latency, make sure both original and alternative designs are set. Simply do a normal get delay on any ancestral node of the transformed node or path delay that goes through this node and you will be prompted with the following dialog.



Enter how often the alternative design will be used as a percentage of total execution. This value should be obtained from software profiling. In the previous $\frac{1}{1+x}$ to $1-x$ example, assume we apply the transform whenever $|x| < 0.5$. The smaller the average absolute values of x are in the input dataset, the higher that percentage will be. The returned value will take into account this transformation effect.

5) There is a built-in polynomial pattern matching rule that transforms $(1+ax)^n$ into $1+anx$ when $|x|$ is below a certain threshold (the threshold has no role in HDG, but rather a software knob that can be tuned during profiling). It automatically searches the entire HDG and finds any structure that represents $(1+ax)^n$. Then it changes the structure to $1+anx$ and set the original and alternative designs accordingly. In the future, we will have more and more built-in pattern matching rules to facilitate design exploration.

11. Naming issues

The name of node is used to suggest its structure. As a result, nodes with different names should be assigned different names. It is basically a TYPE property.

When you load a component into a project several times, you get several copies of the same component and they function exactly the same way. Question is: what happens when you try to modify one of the copies? For example, you want to add an argument to a function, or you want to try a different implementation of an adder. You make that happen by making changes to an existing component (either from standard library or project library). Now you have made your changes and want to navigate away from the component view, but since its structure has been altered, you will be prompted to give it a new name when you leave that level. Here you have options:

1) Keep the old name unchanged. That is a **global update**, which means all other nodes with the same name will also adopt this new structure.

2) Pick a new name. This is a **local update**, which only affects this instance of the node.

An example can help you understand the difference:

We have two composite nodes with same structure (hence same name “new_block1”).

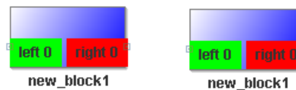


Figure 32: Two nodes with the same structure and name

Go down one of them and change its structure, keep the same name when coming back up. You will find that both new_block1 now have the same new structure. However, if you gave the modified new_block1 some other name, say “new_block_changed”, then the other new_block1 will remain its old structure.

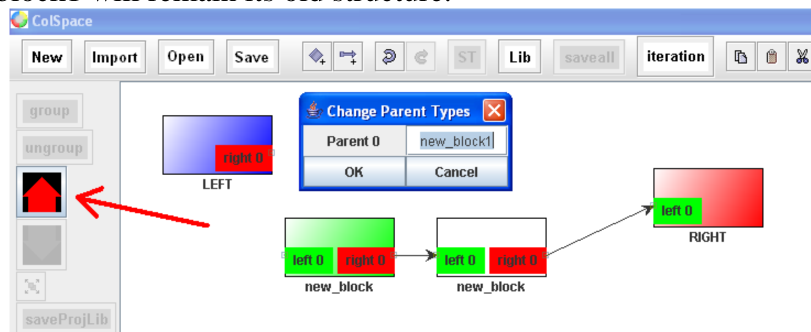


Figure 33: Change parent types

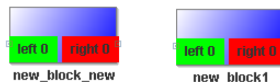
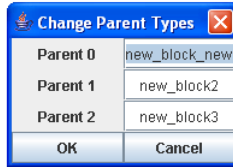


Figure 34: Local update

Nested structures (a component has the same name as its ancestor) are not allowed in ColSpace, because it will confuse ColSpace’s interpretation of its structure. So whenever you insert a node with the same name as the node containing it, you will be prompted to change its name. The same rule applies to grouping too. An example is shown near the beginning of Case Study. For this reason, recursive functions must be

structured in a different way (e.g. tail recursion transformed to simple iteration) to work in ColSpace and indeed in HW implementation in general.

Sometimes you change a node deep in the hierarchy, so its parent is also a child of some other node. In such cases, all ancestors will be listed in the "Change Parent Types" dialog, starting with the most immediate parent on the top. Each parent is subject to the local/global update rules.

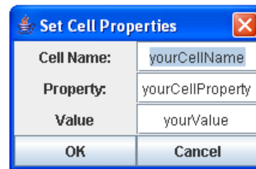


12. Performance

When you create a multi-hierarchy HDG which have thousands of nodes, scalability becomes an issue. Many operations take longer to complete, such as get delay operations. For example, a get delay on the bottom level typically takes no more than an instant while a get delay on the 6th or 7th level takes roughly 10 seconds. A simple way to improve the get delay performance is to cache the latency information. Every node in the HDT has a *DelayTainted* (DT) flag. It is an indicator of whether its stored delay value is fresh or stale information. When you issue get delay on a node for the first time, it will go all the way down to the bottom level to do the calculation and it normally takes a great deal of time. This type of down-to-bottom delay calculation refreshes the delay information of all the nodes evaluated during the process (all the descendents of the top node), and their DT flags will be marked FALSE. Next time you do a get delay on a node with FLASE DT flag, the cached delay value will be returned directly without going down the hierarchy, resulting in a much faster response. When the user changes the delay value of a node, or makes a change to the structure of HDG, all of the ancestral nodes' DT flags will be set to TRUE, meaning that their cached delay values are outdated. A get delay on a node with DT flag marked TRUE will trigger a down-to-bottom delay calculation.

13. Simulation and analysis tools

Under the "Analysis" context menu, there is an option "**change (globally) parameters**". This allows you to change the metrics associated with a TYPE of nodes. The metrics you can change include "latency", "concurrency", "iteration" and "type". The most common usage is to change the latency of a type of node, for example, to change the latency of all the adders from 1 to 2, probably due to adoption of new adder architecture or doubling the width of the adder. You can also do a simulation by varying the latency of a certain type of nodes to see its impact on global latency. This automatic parameter sweeping capability will be added soon.



All you need to do is to enter the TYPE name in the "Cell Name" field, the metric you want to change in the "Property" field (choose one of the above four metrics) and the new value to assume.

"print component contribution/sensitivity" helps you understand what type of nodes make up the majority of the critical path latency. The output looks like:

```
ADD:25.0% MUL:20.0% DIV:45.0% …
```

The values represent both contribution and sensitivity, because we define sensitivity as the change of critical path latency as a result of changing the latency of a particular type of node by 1. So the sensitivity is proportional to the contribution. The contribution/sensitivity data can guide a designer in selecting bottleneck components to apply optimization techniques.

"print all cell types" simply gives an inventory of all different types of nodes present in the HDG. Their respective latencies and the number of instances are also listed.

14. Known bugs

Library components are currently not modifiable.

Get Delay Recursive and **Get Total Delay Recursive** do not work correctly on iteration nodes. This is not essentially a bug because we still have disagreement on their definitions.

15. Features not yet available (but could be expected)

Delete library components.

Pipelines.

The “Import” button for opening a raw-formatted project file is no long supported due to introducing the new port interface feature. Dot is not capable of generating the additional information needed. So please always use standard project file format. I know sometimes it is easier to write a raw-formatted graph description file and have dot add geometric information. It is an interesting and useful topic for future work.

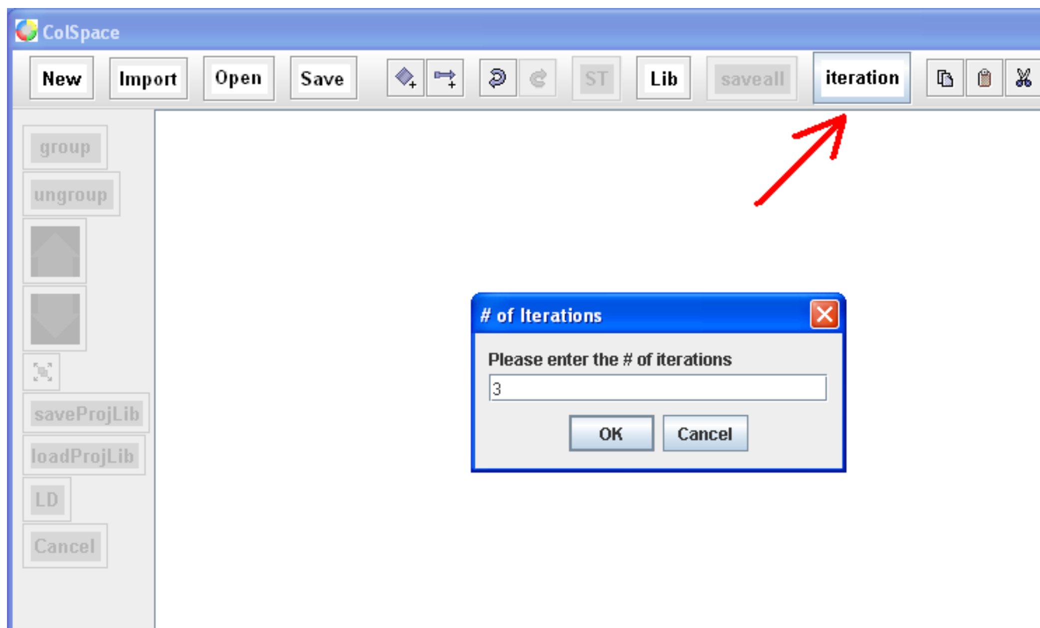
16. Case Study

In this section, I will take a common algorithm (Bubble Sort) and implement it using our ColSpace. It is provided to familiarize you with the basic operations and techniques needed to implement your own algorithm. You will also see how ColSpace helps explore design options and make better design choices for engineers.

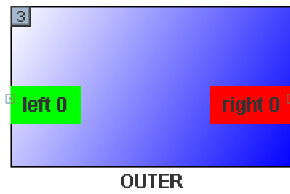
The original C code for bubble sorting an array of 4 elements is listed below. It is a little improved than the straight bubble sort in which swapping is continued until no swap is possible. Since each round of swapping will secure the place of the biggest or smallest element (bubble floating to the surface), the next round of swapping only needs to process one less elements. The total number of comparisons is 3+2+1=6.

```
for (int i=2; i>=0; i++)
{
        for (int j=0; j<=i; j++)
        {
                if (A[j+1] > A[j])
                        swap(A[j],A[j+1]);
        }
}
```

We first create an iteration node and set its # of iterations to 3.

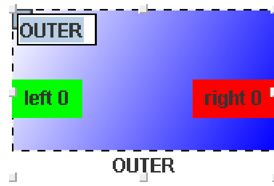


Notice the number of iterations appear on the upper left corner. Double click the node to rename it as “OUTER”.

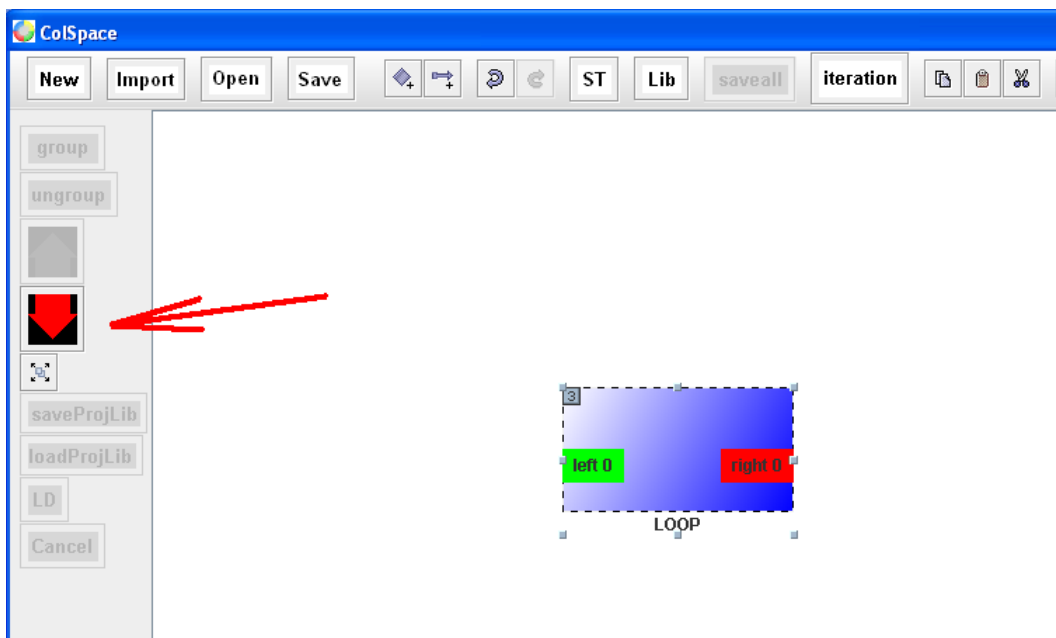


Since we already know the aggregate number of iterations needed, we can simply discard the inner loop and assume the outer loop to iterate 6 times. It is simple loop transformation, which does not have impact on the performance (latency) analysis. Change the number of iterations to 6 by selecting from the popup menu and also change the name “OUTER” to “LOOP” by double clicking.

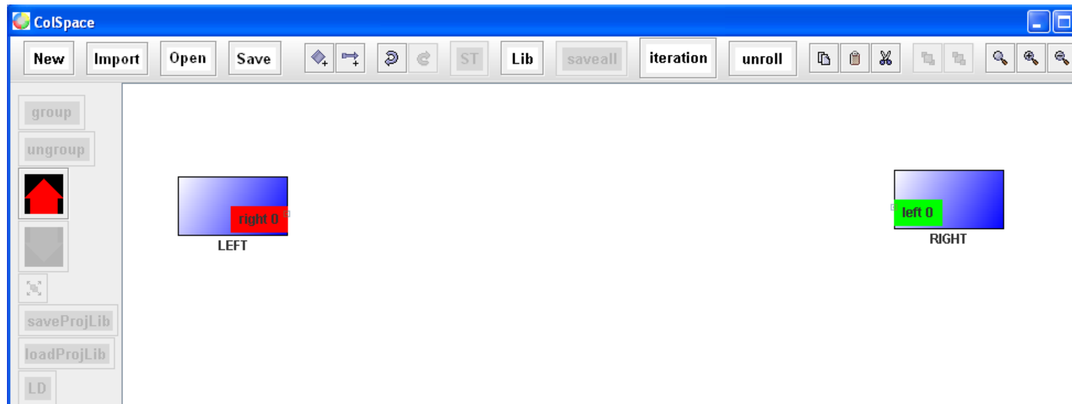
--->cell1<---
Edit
Remove
Insert
Set Delay
Get Delay
Get Total Delay
Get Delay_Recursive
Get Total_Delay_Recursive
Get Critical Path
Get Critical_Path_Recursive
Get Critical_Path_Bottom
Duplicate
Set numLeftPorts
Set numRightPorts
SET LEFT PORTS
SET RIGHT PORTS
Exploit Parallelism
Set # of inputs consumed/iter
Set # of outputs produced/iter
Set # of Iterations
Get Location
Get All Delays in Current View
Get All Delays in the Entire Proj
Depend On
Clear HighLights



Click on the downward arrow button to navigate inside LOOP.



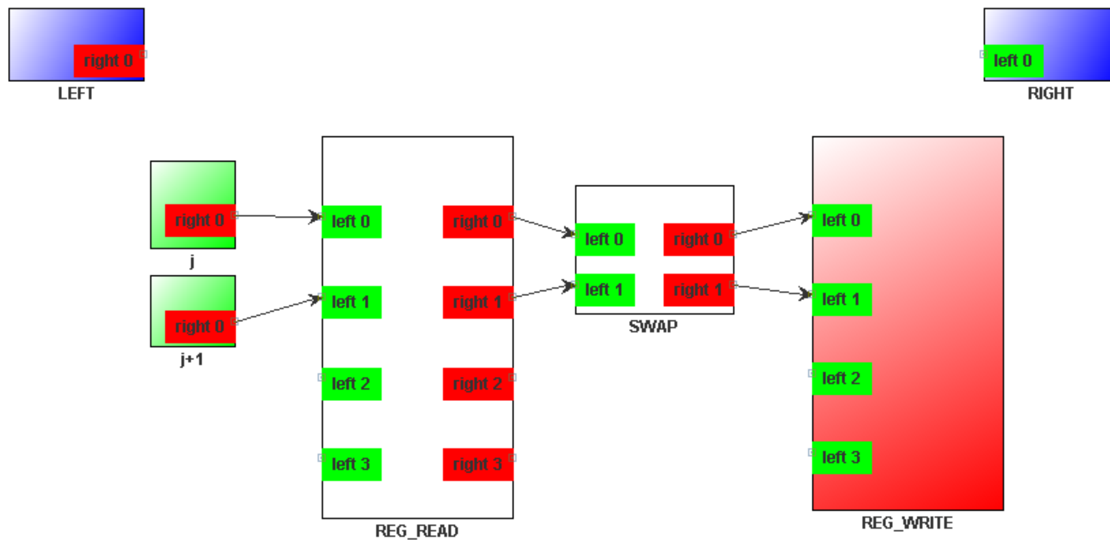
Here you see two **Port Cells**. You can add various components in this view and connect them with the two port cells to complete the underlying structure of LOOP. Be careful about the naming because nested hierarchy (one TYPE of a node containing another instance of that TYPE as its child) is not allowed. In case of a child node sharing the same name as one of its ancestor, user will be prompted to change its name.



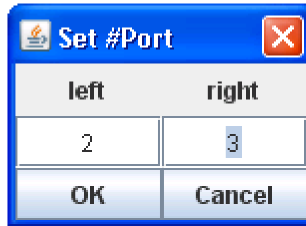
Here's the structure I created for bubble sort of 4 numbers. Numbers are stored in a register. Every iteration, two indices are generated to retrieve two numbers from the REG, and these two numbers are fed into a functional block called SWAP. The SWAP basically achieves the following code segment:

```
if (A[j+1] > A[j])
    swap(A[j],A[j+1]);
```

Inputs are compared and swapped to output if not in order or preserved to output if already in order. In this example it's OK to combine the two operations into a whole entity because the comparison and swapping are always executed consecutively. The two outputs are in turn written back into the REG in the position pointed to by the two indices and waiting for next round of swapping.

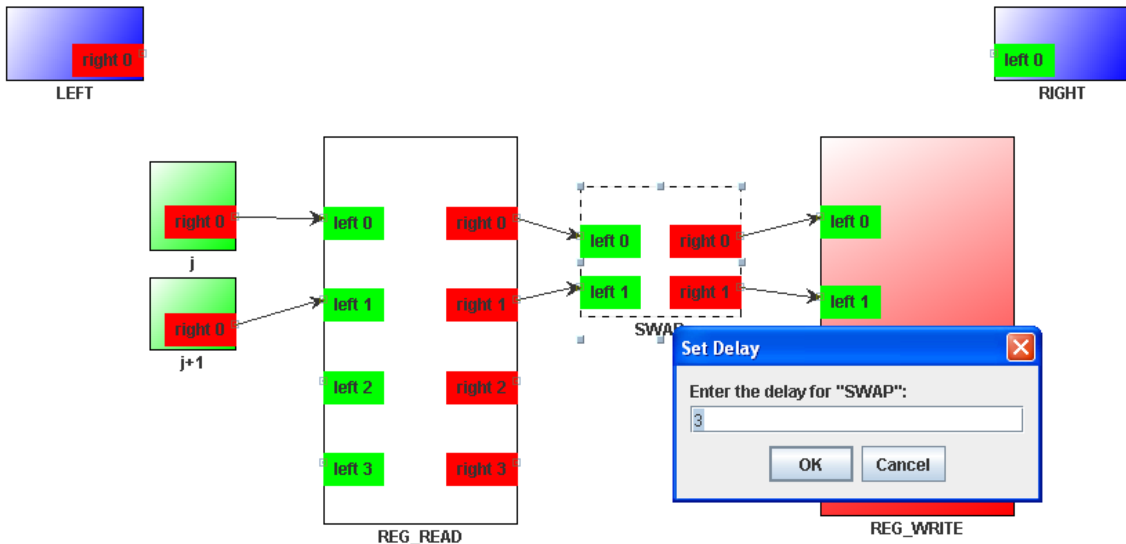


Make use of "Set #Ports" and "Set Ports" from the right click context menu to change port names and quantities.

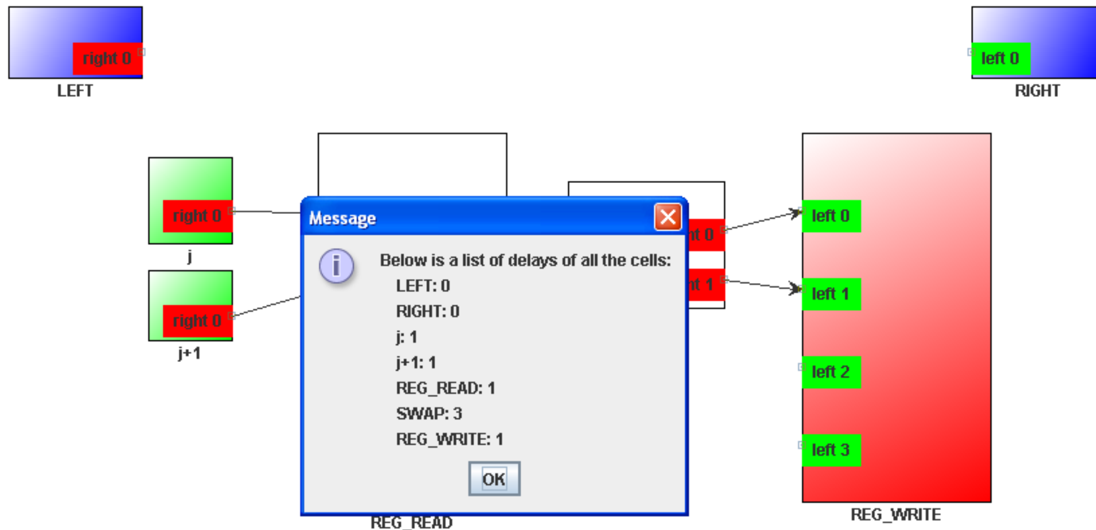


Also note that green indicates a data source, red a data sink, blue an isolated node and white a fully connected node.

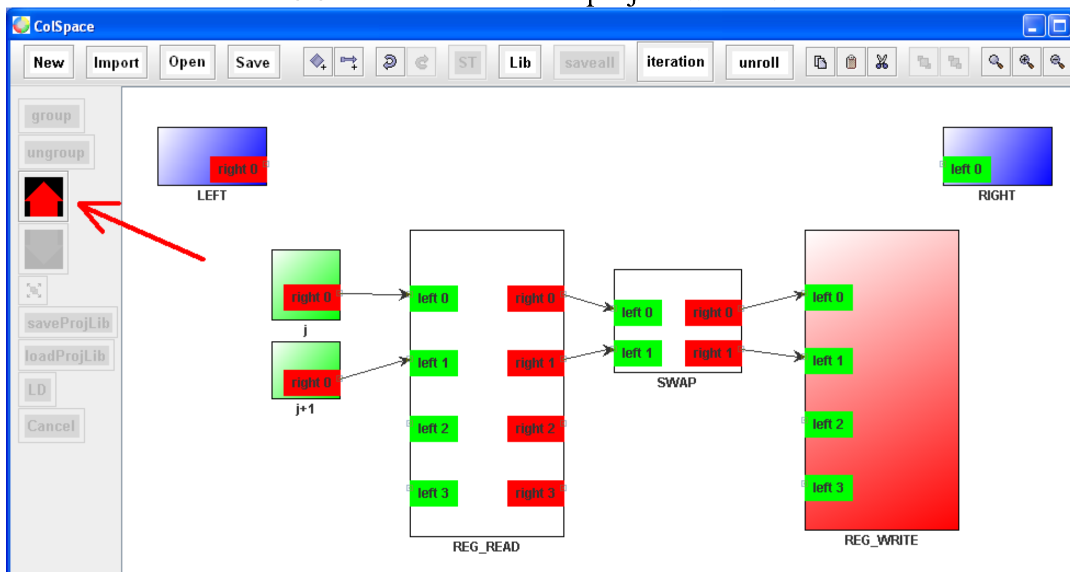
Since the swapping is the dominant operation in sorting, we assign it a latency value of 3 while the rest of functional blocks remained with latency 1. You can change latency by right clicking on a node and choose “Set Delay”.

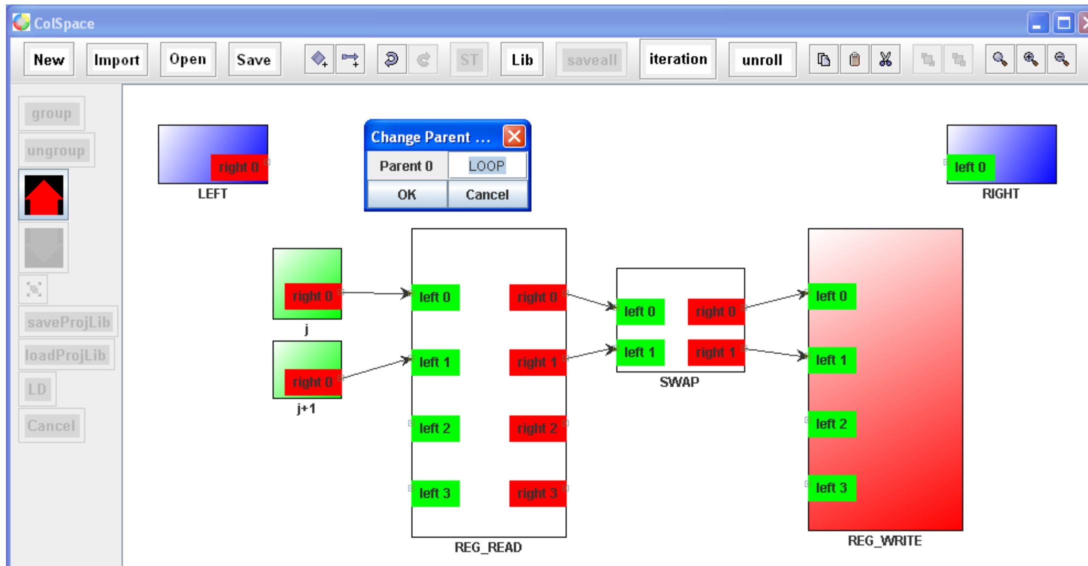


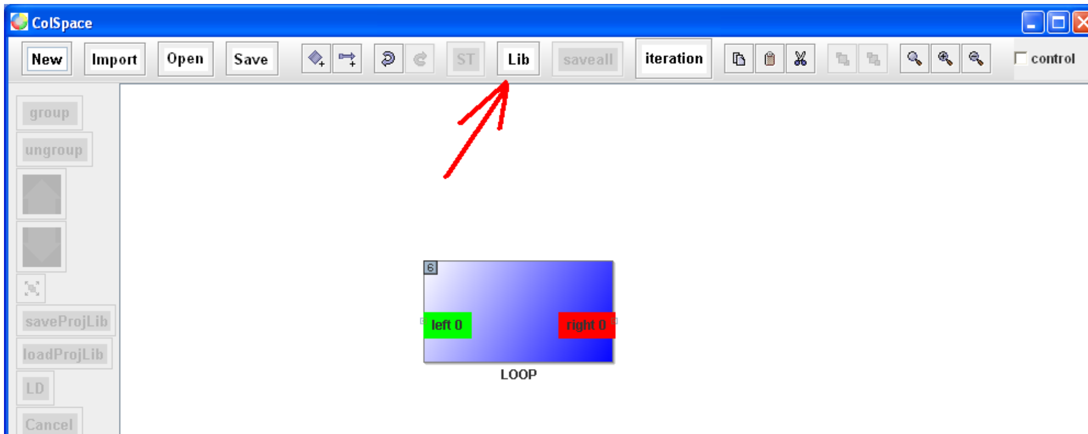
We are done with the initial HDG design four our algorithm. So far so good. Let’s overview latency information by right clicking on a blank region and choose “Get All Delays in Current View”. You should see something like this (port cells have delay of zero because they are just virtual connectors):

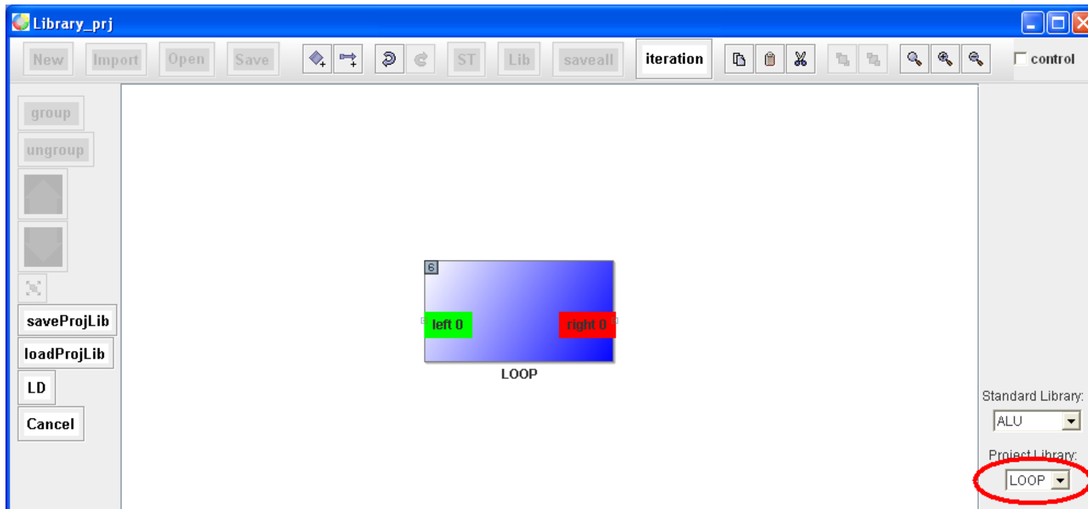


Then we navigate up by clicking on the upward arrow button. This will take you back to the LOOP level. However, you might be prompted to rename the LOOP construct. This is because you have just changed the underlying structure of LOOP and you need to make a decision whether to update all LOOPs (previous and following) to this new structure or only to give this LOOP a different TYPE. Since we are not creating any more loops, simply click on OK to save this structure. Click on the "Lib" button to enter the library view. Notice that LOOP has just been added to our project library. Later on, we can choose to load this LOOP structure into our project whenever needed.

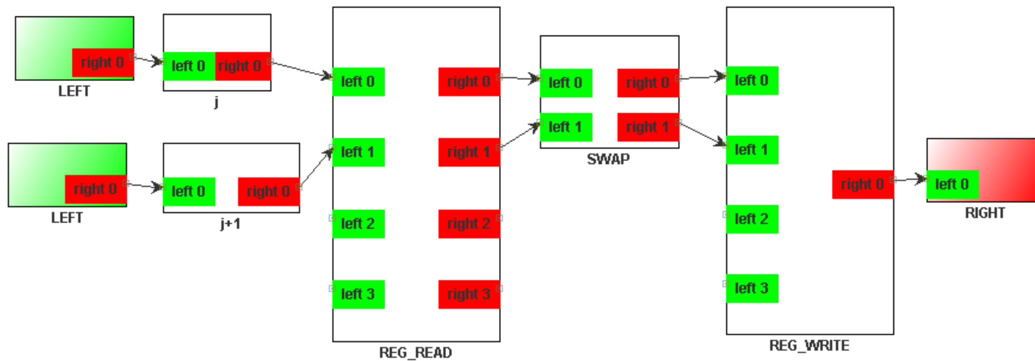




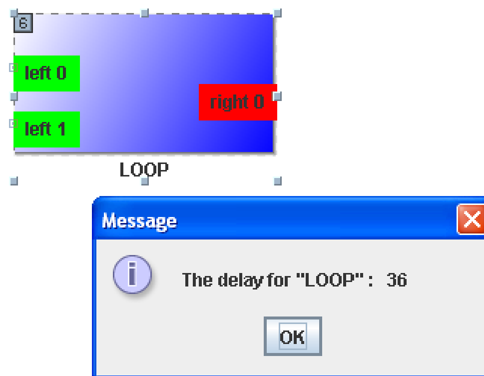




Add a left port to LOOP and properly connect its port cells with its underlying children.



Finally, right click on LOOP and choose "Get Delay". It shows 36, which is the right answer (6 iterations * (1+1+3+1)/iteration). Remember, you need to properly connect all of the port cells of a group node in order for its "Get Delay" to work. It basically calculates the delay between input ports and output ports.



Now, we have a working HDG of bubble sort, with total system latency 36 and no optimization applied. Let's save this project for latter contrast before we move on to some optimization techniques. Click on "Save" button and save it in a certain location in your file system. Next time you can open the same project by choosing "Open".

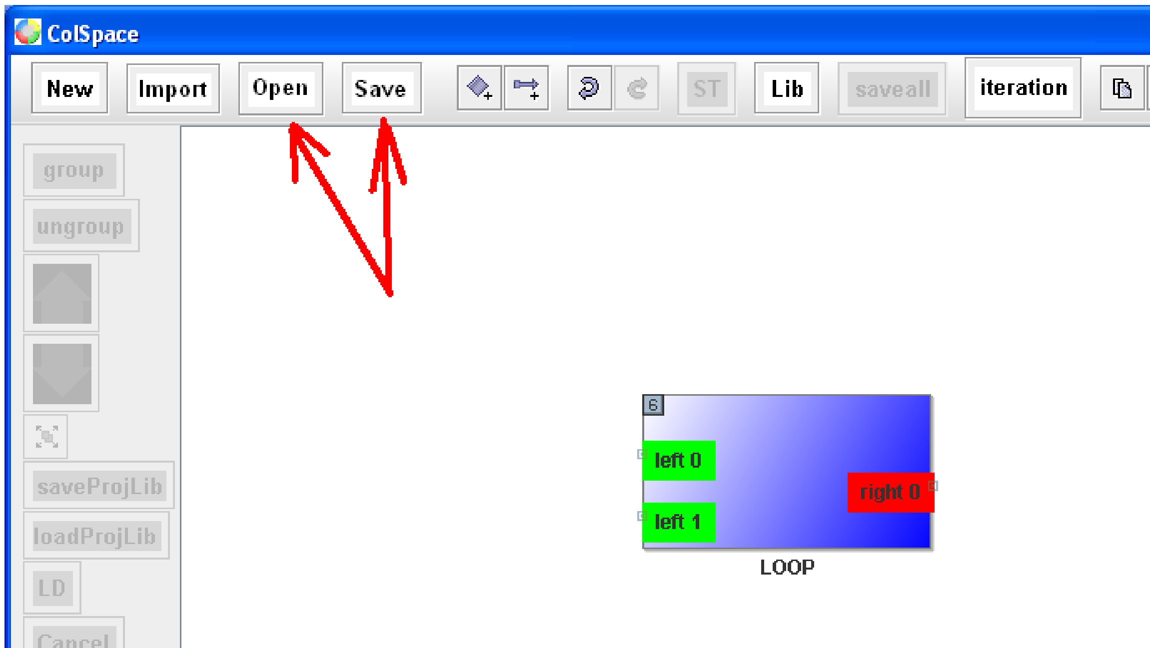
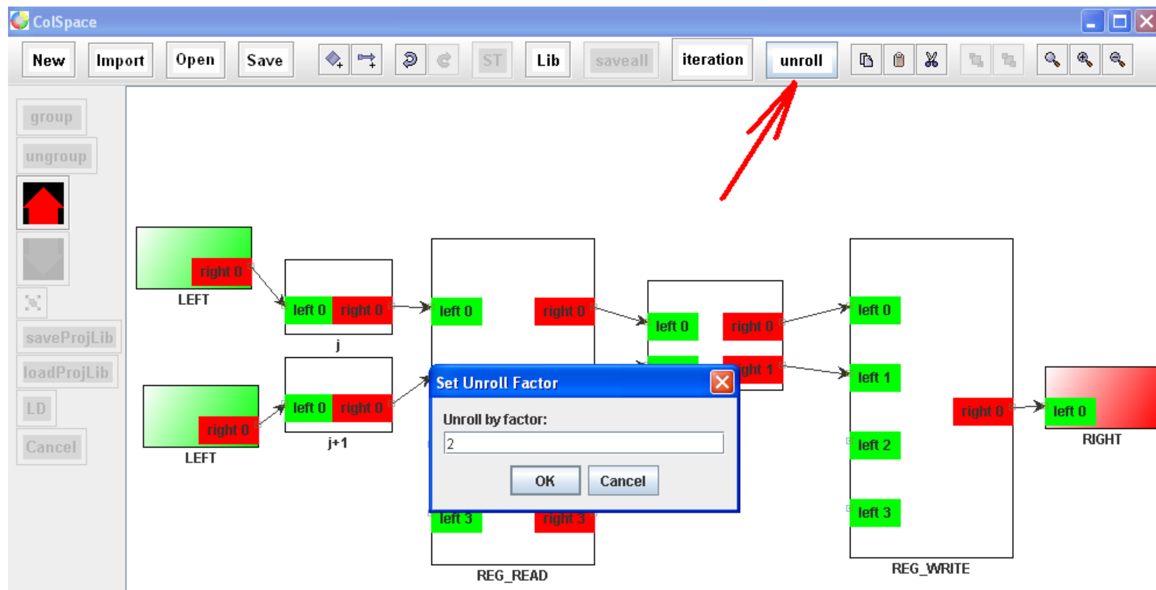


Figure 35: Saving and loading the project

Simply by looking at the loop body, there's not much we can do. So, let's try loop unrolling and see whether optimization possibility will open up. Go inside the LOOP and click on "unroll" button (located in the top toolbar). You will be asked to provide the **unrolling factor**: the number of copies of loop body that you intend to produce for each loop body as current, and the number of iterations will accordingly be divided by that factor. For example, unrolling by factor of 3 will produce additional two copies inside the loop body (making totally 3) and the total number of iterations will be reduced to its one-third. For this example, use 2.



Another identical copy is created. Initially the duplicate might not be placed in a suitable position, so feel free to drag it around to suit your need. You should be able to reach the following configuration:

This HDG is not accurate. It requires some manual tweaking to expose inter-iteration dependencies. First of all, one of the j or j+1 should be removed and the other changed to j+2. REG_READ and REG_WRITE can be shared. It's easy to delete nodes or edges, just click and hit "delete" on your keyboard. Port Cells are not removable, but rather have to be removed through "set #Ports".

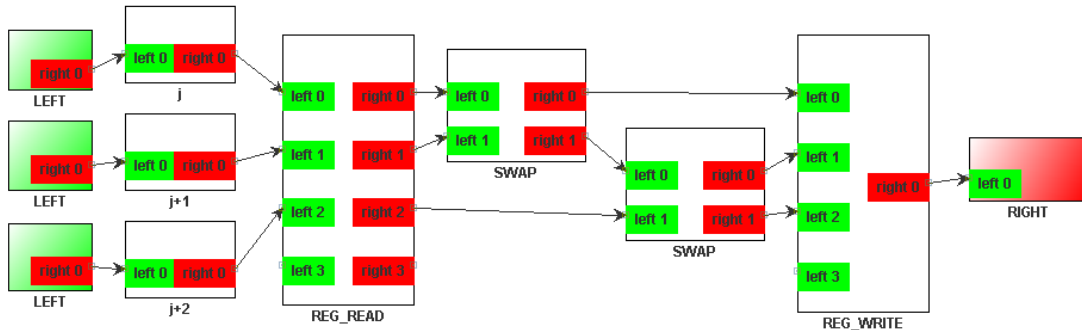
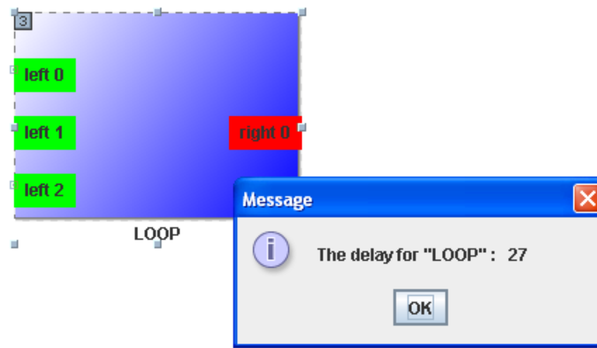


Figure 36: Bubble sort after first unrolling and dependency correction

Now let's try a different way of calculating delay. Right click on REG_WRITE or RIGHT, choose "Get Delay" -> "Get Total Delay". That should give you 9. That is the delay calculated from any of the input (LEFT) up till and including REG_WRITE / RIGHT (RIGHT has latency 0). 1+1+3+3+1 = 9.

And now if you go up one level, click on LOOP and get its delay, you'll get 27, which is 9*3. Our first improvement is achieved!



An easy victory shall not satisfy us. Since it is a nested loop, and we have been mainly focusing on inter-loop dependency among inner loops, we should try our luck on outer-loops. Now, structure our HDG to reflect the three outer loops, each has decreasing number of inner loops. Total delay is 20 (1+3*3+2*3+1*3+1). This reduction in delay is due to extensive use of the above technique, sharing and elimination of registers as a result of inter-inner-loop dependency analysis.

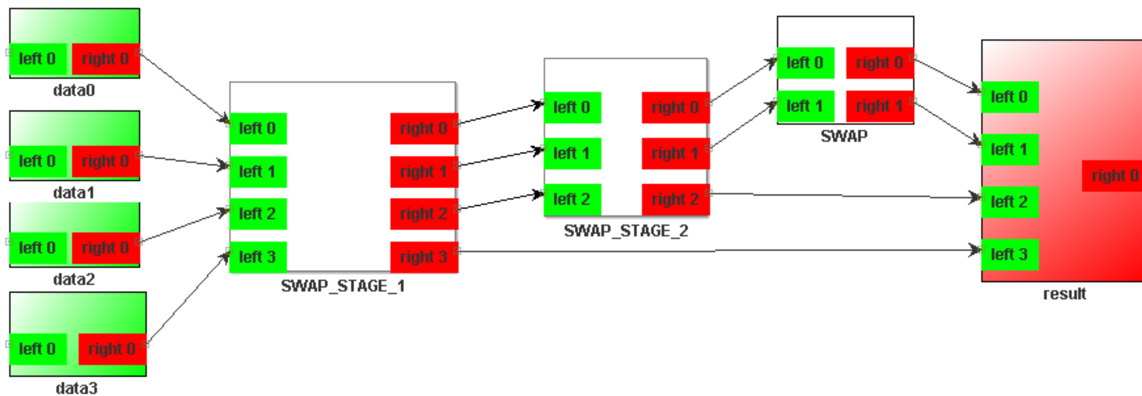


Figure 37: Final structure of bubble sort

Right click on “result” and choose “Exploit Parallelism”.

Duplicate
Set numLeftPorts
Set numRightPorts
SET LEFT PORTS
SET RIGHT PORTS
Exploit Parallelism
Set # of inputs consumed/iter
Set # of outputs produced/iter
Set # of Iterations
Toggle Ordered
Set Concurrency
Toggle Pipeline On/Off
Get Location
Get All Delays in Current View

You will see three edges connecting SWAP_STAGE_1 and 2 get highlighted. This indicates that there is opportunity to rearrange the underlying nodes of SWAP_STAGE_1 and 2 to reduce the overall latency. This opportunity is identified by a mismatch of total delay and total delay recursive.

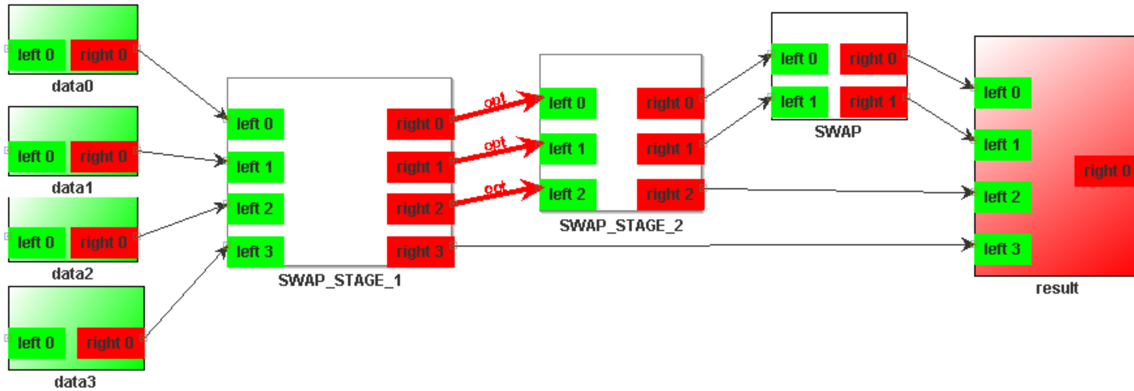


Figure 38: Hidden parallelism highlighted

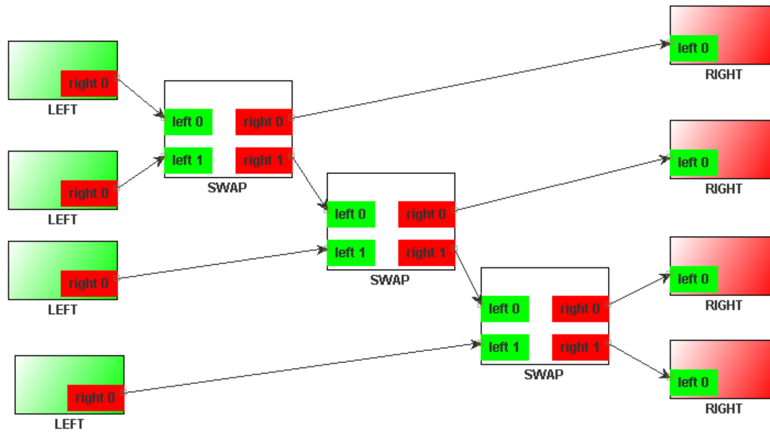


Figure 39: SWAP_STAGE_1

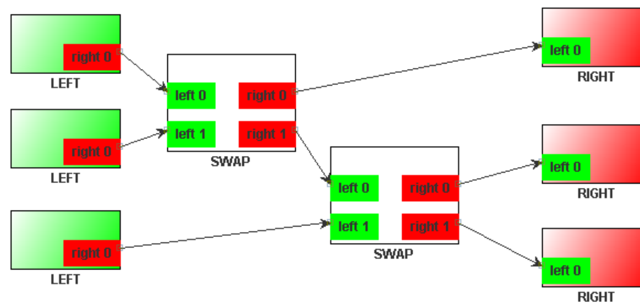
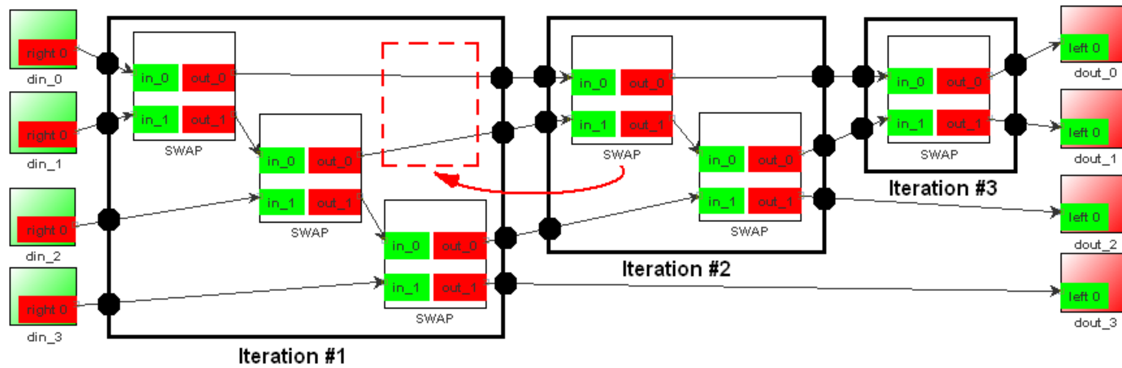


Figure 40: SWAP_STAGE_2

See how the optimization is achieved:



The first SWAP of Iteration #2 can potentially start execution before the last SWAP in Iteration #1 completes, leading to a one time step reduction in total latency. More delay savings can be obtained when sorting a large number of inputs. Notice that, by taking advantage of this implicit algorithm, we will be performing two SWAPs at the same time, requiring double hardware resources (previously only one SWAP needed at a time). Hardware and software engineers can assign different weight to area and performance to direct ColSpace make the best tradeoff decision.

For your convenience, the finished bubble sort project is included in the \example directory.

17. c2colspace – creating ColSpace project from C source code

There is a /tools subdirectory in your downloaded package. Under that you will find two Java applications: *vcg2dot* and *dot2colspace*. Together with a compiler called *scale*, you can automatically create a ColSpace project directly from C source code you probably already possess.

The running instructions are included in the readme file under the same directory. For convenience, they are repeated here:

Step 1

Download Scale from http://www-ali.cs.umass.edu/Scale/download2.html. Scale is a C compiler which we will use to generate intermediate graph representations for use with ColSpace. Please choose the "Scale jar file".

Step 2:

We should first generator a "vcg" file from the C source. More information about "vcg" format can be found at http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html. Scale has some HW requirements. I was able to run it on adder and mamba machines, but not power*. EE students can use roseriver. To do, set CLASSPATH to include the scale.jar you just downloaded. For example, if scale.jar is located under a /scale directory. Go to that directory and issue command (if you are using BASH):

```
export CLASSPATH=scale.jar:$CLASSPATH
```

Then, you can run scale using the following command:

```
java scale.test.Scale -scribble_graph_before "0" -vcg mycpp.cpp
```

Or combile the two:

```
java -classpath scale.jar scale.test.Scale -scribble_graph_before "0"
    -vcg mycpp.cpp
```

Replace mycpp.cpp with the name of your source file. Each function in the source will produce a vcg file. "0" means no optimization will be applied.

Step 3

If you want to view the vcg graph, there are several options. The graph will be displayed on your terminal (assume you have XWindows support) if you omit -scale option above. You can also use some tools mentioned in http://www.xs4all.nl/~twlevo/ and http://code.google.com/p/vcgviewer/. For Windows user, I recommend vcgviewer.

Step 4

Run vdg2dot: vcg2dot vcgfile dotfile

Step 5

Install Graphviz: http://www.graphviz.org/. Launch dot: Graphviz -> dot. Choose the generated dotfile as input. Specify your desired output file and choose dot as output type. Click "Do layout". This will generate a dot file that has the geometric information.

Step 6
Run dot2colspace: dot2colspace geometric_dotfile colspace_proj_file

Step 7
Open the colspace_proj_file in ColSpace :)

****

AUTHOR: Jiawei (Jarvis) Huang
jh3wn@virginia.edu

Let's make ColSpace better!

Report bugs and make suggestions to jh3wn@virginia.edu.