

Subscheduling

Introduction

A **subschedule** is used to reduce the granularity of primitive execution and, as a result, reduces memory use. Subscheduling allows a set of primitives that would have run at a high granularity to be executed as a group at a lower granularity. The subschedule must be run repeatedly by the higher-level schedule so that the total number of firings of the primitives in the subschedule is not changed. For example, a primitive that would have to fire at a granularity of 100 in the top-level schedule could run at a granularity of 10 in a subschedule if the subschedule is fired 10 times by the higher-level schedule.

Frequently primitives running in the same static schedule execute at different granularities. For example, a graph that contains an `m_v` primitive that converts 1 $R \times C$ matrix into R vectors may have all the matrix processing execute at a granularity of 1, while the vector processing operates at a granularity of R . The default schedule generated for this graph will call each vector box at a granularity of R . We say that the natural granularity of the vector primitives in this graph is R because they must run at this granularity to complete one cycle of the static schedule. A static schedule running this graph might execute the primitives in the following order:

```
matrix box1 @ granularity 1
matrix box2 @ granularity 1
m_v box     @ granularity 1
vector box1 @ granularity R
vector box2 @ granularity R
...
vector boxn @ granularity R
```

Subscheduling can be used to change this order to:

```
matrix box1 @ granularity 1
matrix box2 @ granularity 1
m_v box     @ granularity 1
for (i = 1..R) {
  vector box1 @ granularity 1
  vector box2 @ granularity 1
  ...
  vector boxn @ granularity 1
}
```

In the subschedule, instead of running each vector box at its natural granularity of R , all the vector boxes are run at a granularity of 1 and this process is repeated R times.

Two advantages result from using subschedules. First, the vector boxes can operate in a smaller amount of memory, which means, that cache or other fast but small memory resources can be used to process the data. This technique of firing a set of algorithms repeatedly at a smaller granularity is commonly known as **strip mining** and can result in significant performance improvements. Secondly, in examples that follow we will show how for some applications overall memory usage can be reduced to make larger applications fit into memory.

Using some of the Gedae tools, the user can implement subscheduling by putting all of the vector processing boxes into a common subschedule. Gedae then creates two schedules – the top-level schedule that calls the subschedule and the lower-level schedule. If all the vector boxes in the above example were marked to run in the same subschedule, then Gedae would create two schedules as follows:

Top-level Schedule:

```
matrix box1 @ granularity 1
matrix box2 @ granularity 1
m_s box     @ granularity 1
subsched    @ granularity R (Runs Subschedule 1 below R times)
```

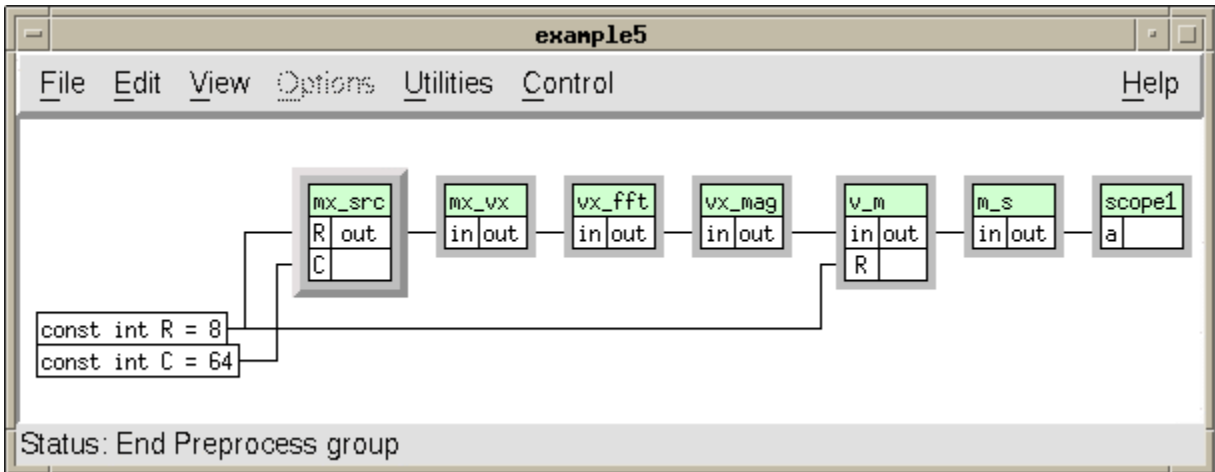
Subschedule 1:

```
subdeq      @ granularity 1
vector box1 @ granularity 1
vector box2 @ granularity 1
...
vector boxn @ granularity 1
```

Data is moved in and out of the subschedules at the subdeq and subenq interface boxes that are auto-generated into the subschedule. In reality the subenq/subdeq box do nothing. Instead, pointers in the boxes immediately following the subdeq and preceding the subenq are manipulated by the subsched box called from parent schedule before each firing of the subschedule. So the beginning and end boxes in the subschedule operate directly out of the parent schedules memory without copying any data.

An Example

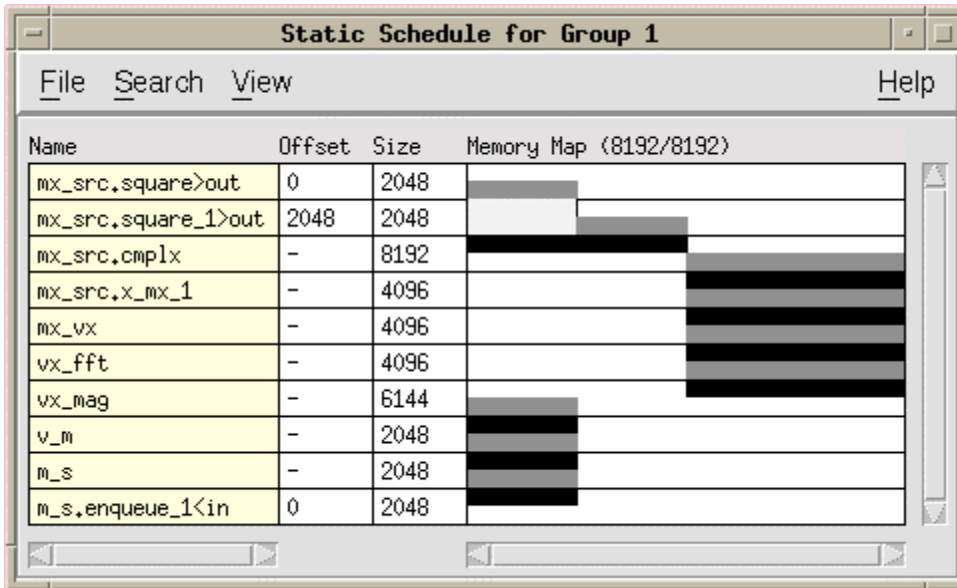
Consider the following graph:



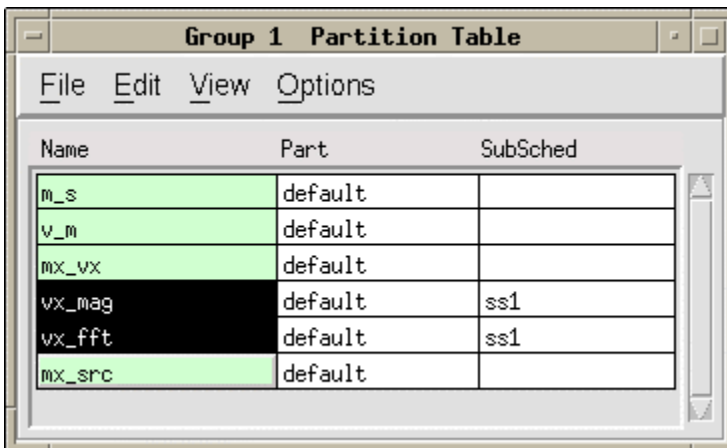
Because the matrix size is 8x64, the `vx_fft` and `vx_mag` boxes will need to fire with a granularity of 8. The granularity of 8 is necessary because for each matrix processed the `mx_vx` box produces 8 vectors to be processed. The granularities of the `vx_fft` and `vx_mag` box are viewable in the granularity table below:

Name	Gran Mult	Nat Gran	Max Gran	Granularity	Priority
Schedule 1	1				
m_s		1	1		0
enqueue_1		512	512		0
mx_src					
cmplx		512	512		0
square		512	512		0
square_1		512	512		0
x_mx_1		1	1		0
mx_vx		1	1		0
v_m		1	1		0
vx_fft		8	8		0
vx_mag		8	8		0

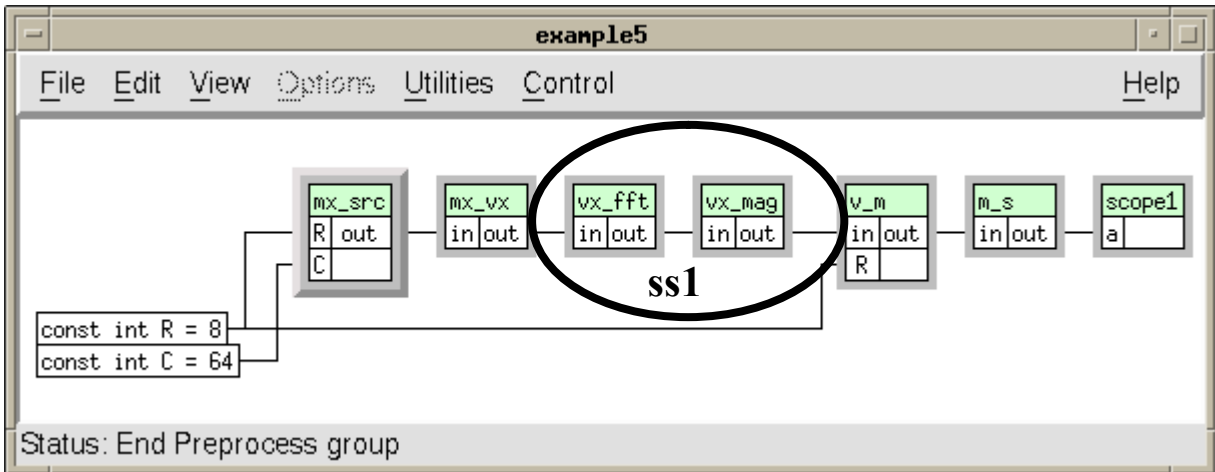
The static schedule that executes this schedule is seen below. The static schedule display shows the firing order of the boxes vertically and the memory usage of the boxes horizontally. The display shows that the `vx_fft` and `vx_mag` boxes directly process all of the vectors provided by a single firing of the `mx_vx` box.



To create a subschedule for the vx_fft and vx_mag boxes, the user sets the subschedule field in the Partition Table dialog to some name – in this case, ss1.



As a result of marking these boxes, s being part of subschedule ss1, the vx_mag and vx_fft boxes circled in the graph below are replaced by a subsched box. The subsched box runs the vx_fft and vx_mag boxes with a granularity of 1 instead of 8.



This modification of the graph can be viewed in the Granularity Table:

Name	Gran Mult	Nat Gran	Max Gran	Granularity	Priority
Schedule 1	1				
SubSched ss1	1	8	8		0
vx_fft	1	1	1		0
subdeq_1	1	1	1		0
vx_mag	1	1	1		0
subenq_1	1	1	1		0
m_s		1	1		0
enqueue_1		512	512		0
mx_src					
cmplx		512	512		0
square		512	512		0
square_1		512	512		0
x_mx_1		1	1		0
mx_vx		1	1		0
v_m		1	1		0

In the table, the Name field shows a hierarchy of schedules, subschedules and primitives. Any subschedule appearing in a higher-level schedule or subschedule acts like a primitive within that schedule. The Natural Granularity field (Nat Gran) is the granularity that the box must fire at to complete one cycle of the schedule in its immediate parent. The Granularity Multiplier (Gran Mult) is a user settable field that allows the user to multiply the natural granularities of primitives and subschedules immediately contained within a schedule. The user can set the Gran Mult field to achieve higher granularities than the natural granularity, thereby, reducing the overhead required to fire a primitive. Setting the Gran Mult field provides the opposite effect of subscheduling. The user has complete control over the granularity – to either raise it by setting the granularity multiplier or lower it by marking boxes to be subschedule. The Maximum Granularity (Max Gran) is

the granularity that the primitives will fire at given their natural granularity and granularity multiplier. Thus,

$$\text{Max Gran} = \text{Gran Mult} * \text{Nat Gran}$$

The table shows that the vx_fft and vx_mag boxes are replaced by the Subsched ss1 box in the Schedule 1. The subschedule must fire 8 times in the higher-level schedule so that the total number of times the vx_fft and vx_mag boxes fire remains 8. The top-level schedule seen below replaces the vx_fft and vx_mag boxes with a single call to a subschedule box.

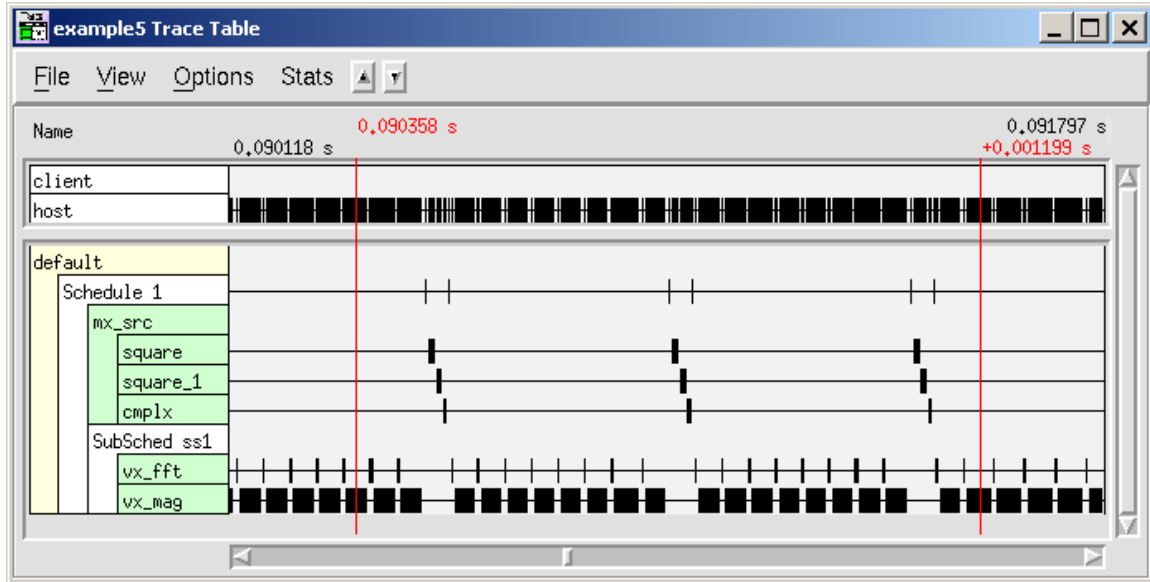
Name	Offset	Size	Memory Map (8192/8192)
mx_src.square>out	0	2048	[0, 2048]
mx_src.square_1>out	2048	2048	[2048, 4096]
mx_src.cmplx	-	8192	[0, 8192]
mx_src.x_mx_1	-	4096	[4096, 8192]
mx_vx	-	4096	[4096, 8192]
vx_fft.subsched_1(ss1),noop_1	-	4096	[4096, 8192]
vx_fft.subsched_1(ss1)	-	6144	[0, 6144]
v_m.noop_2	-	2048	[0, 2048]
v_m	-	2048	[0, 2048]
m_s	-	2048	[0, 2048]
m_s.enqueue_1<in	0	2048	[0, 2048]

The subschedule box in the above schedule executes the schedule below 8 times.

Name	Offset	Size	Memory Map (768/768)
vx_fft.subdeq_1>out	0	512	[0, 512]
vx_fft	-	512	[0, 512]
vx_mag	-	768	[0, 768]
vx_mag.subenq_1<in	512	256	[512, 768]

As previously noted, the subdeq and subenq entries in the table are the points where data is moved out of parent memory to the subschedule. In actual fact, no data is moved. Rather, the parent schedule subsched box directly modifies the pointer to "in" in the vx_fft box and the pointer to "out" in the vx_mag box. In this way, the pointers at the boundary of the subschedule are modified to point to the correct location in the parent schedule memory.

The subschedule execution can be viewed on the Trace Table below, which shows how the boxes in the subschedule fire multiple times for each firing of the parent schedule boxes:



The above example, while quite simple, illustrates one of the advantages of subscheduling. The `vx_fft` and `vx_mag` boxes execute in a smaller amount of memory. As a result, it is more likely that the data passed from the `vx_fft` to the `vx_mag` box can remain in cache, thereby increasing the performance of the algorithm. If many boxes are in the subschedule, then the benefit is even greater because there will be no cache misses over many box firings.

Changing the Subschedule Granularity

In the above example, the subscheduled primitives run at a granularity of 1. A granularity of 1 produces the smallest schedule size. However, if it is desired, it is possible to set the granularity of the subschedule higher. Increasing the granularity above 1 can reduce the overhead for running the boxes, while still saving memory. The subschedule granularities can be increased by a multiplier anywhere between 1 and the original granularity of the primitive. (Setting the granularity multiplier to the original primitive granularity is the maximum multiplier possible and effectively defeats the purpose of subscheduling. No reduction in granularity results because the subschedule will run with the primitives set at their original granularity).

To set the subschedule granularity, bring up the Firing Granularity Table from the Group control dialog, and select the Subschedule box (in this case `SubSched ss1`) as in the dialog below:

Name	Gran Mult	Nat Gran	Max Gran	Granularity	Priority
Schedule 1	1				
SubSched ss1	1	8	8		0
vx_fft		1	1		0
subdeq_1		1	1		0
vx_mag		1	1		0
subenq_1		1	1		0
m_s		1	1		0
enqueue_1		512	512		0
mx_src					
cplx		512	512		0
square		512	512		0
square_1		512	512		0
x_mx_1		1	1		0
mx_vx		1	1		0
v_m		1	1		0

With the Subsched ss1 box selected, click on the menu item Options->Set Granularity Multiplier and fill in the multiplier for this schedule.

Input Dialog

Set the granularity multiplier of the selected schedules

OK Cancel

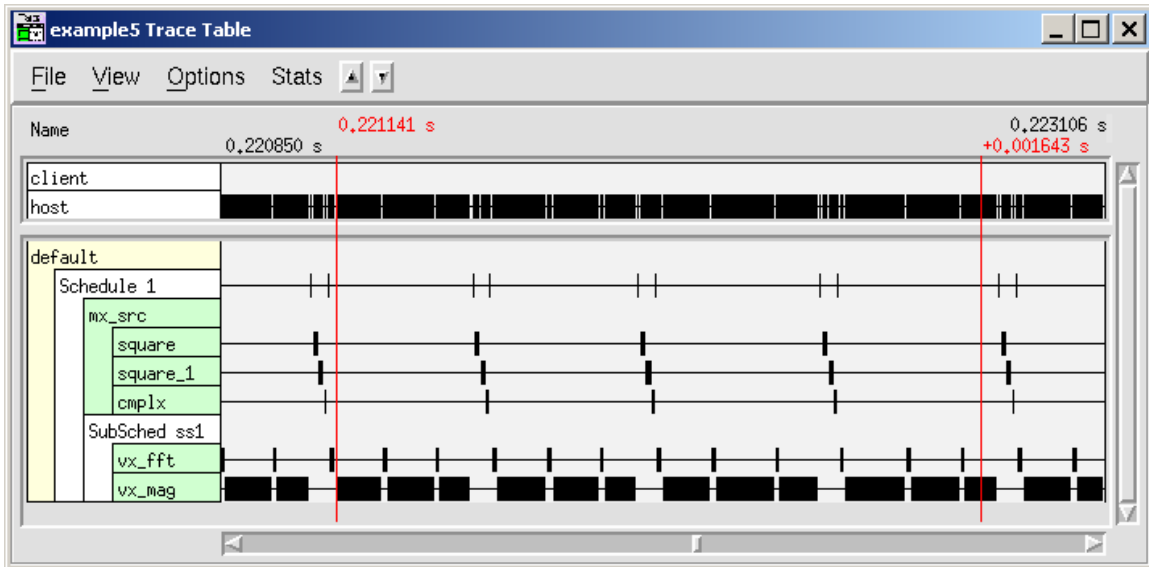
In this case, we set the multiplier to 3.

The screenshot shows a window titled "Group 1 Granularity Table" with a menu bar (File, Edit, View, Options, Help) and a table with the following data:

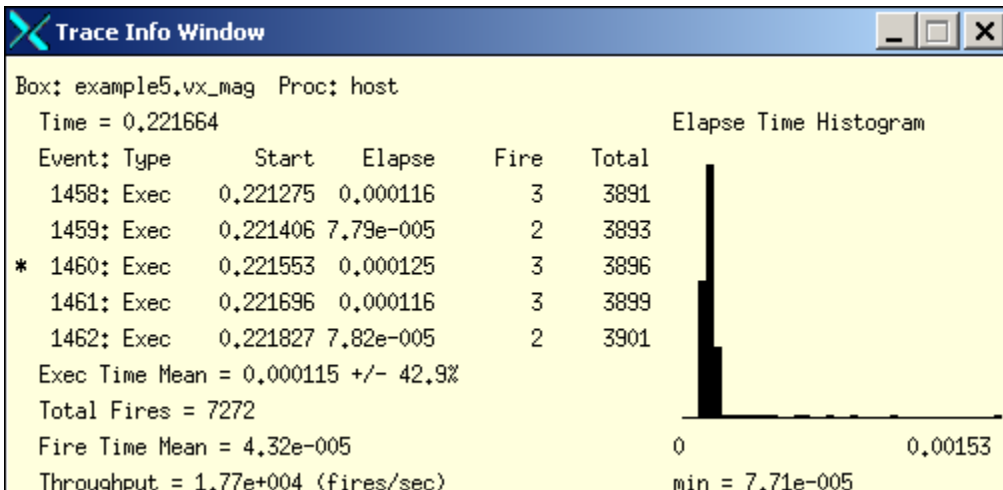
Name	Gran Mult	Nat Gran	Max Gran	Granularity	Priority
Schedule 1	1				
SubSched ss1	3	8	8		0
vx_fft		1	3		0
subdeq_1		1	3		0
vx_mag		1	3		0
subenq_1		1	3		0
m_s		1	1		0
enqueue_1		512	512		0
mx_src					
cplx		512	512		0
square		512	512		0
square_1		512	512		0
x_mx_1		1	1		0
mx_vx		1	1		0
v_m		1	1		0

Note that the multiplier does not have to be an even submultiple of the subschedule's natural granularity of 8. Setting the subschedule granularity means that the subschedule will fire three times with granularities 3, 3 and 2. The subsched primitive will fire the subschedule as many times as possible at the maximum granularity and then flex the schedule's granularity down to handle the remainder of the firings.

The Trace Table that results from setting the SubSched ss1 granularity multiplier to 3 is shown below:



Clicking on the vx_mag box line with mouse button three shows that the box does indeed fire with a repeating pattern of 3, 3, and 2.

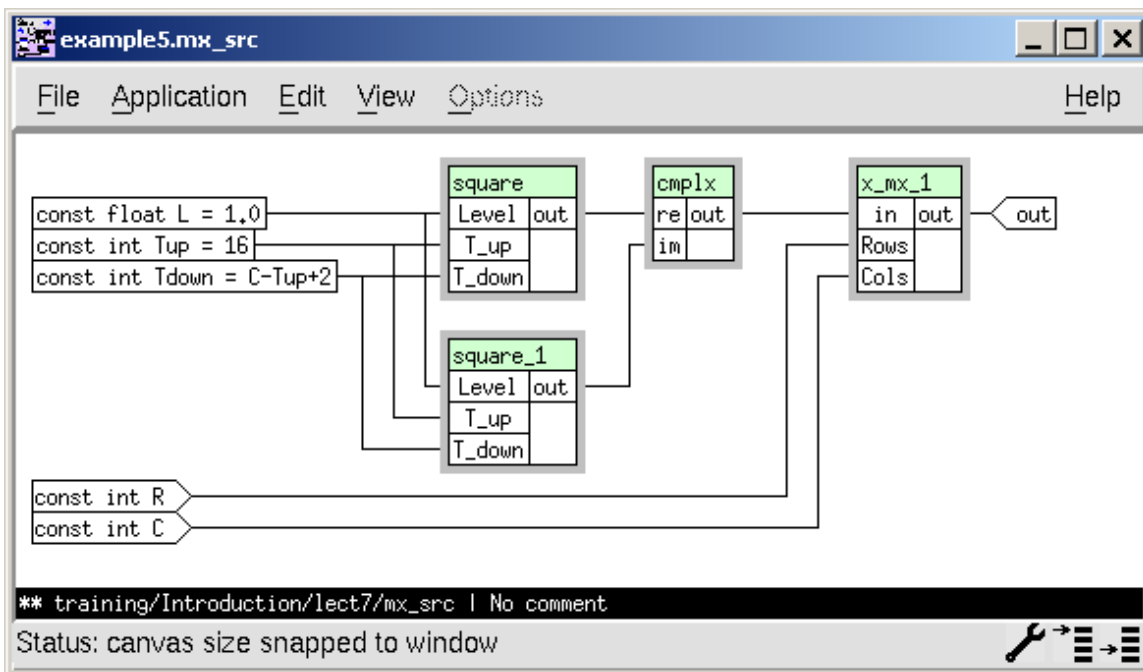


Subschedule Connectivity

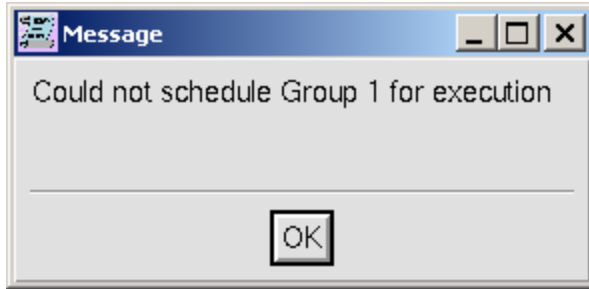
Subschedules must be connected sets. That is, if a group of boxes are all labeled with the same subschedule name, then it must be possible to trace a path between these boxes in the graph without exiting and reentering the subschedule. For example, the following subscheduling would be incorrect:

Name	Part	SubSched
m_s	default	
v_m	default	
mx_vx	default	
vx_mag	default	
vx_fft	default	
mx_src	default	
square_1	default	ss *
square	default	ss *
x_mx_1	default	
cplx	default	

because the square_1 and square boxes are not directly connected, as seen in the mx_src box subgraph below:



When attempting to run this graph, the error manifests itself prior to execution during graph scheduling. The following dialog appears:



and the following terminal output explains the problem:

```
Status: Begin Preprocess group
ERROR: An output of subSchedule [ss] leaves and then
reenters
-----Connected boxes:
  example5.mx_src.square_1
-----Boxes not connected to first group:
  example5.mx_src.square
Error: Preprocessing of group Group 1 failed
```

To correctly create a subschedule with these boxes, the `cmplx` box must be included in the subschedule; then the boxes in the subschedule are connected, and the subschedule will be correctly created.

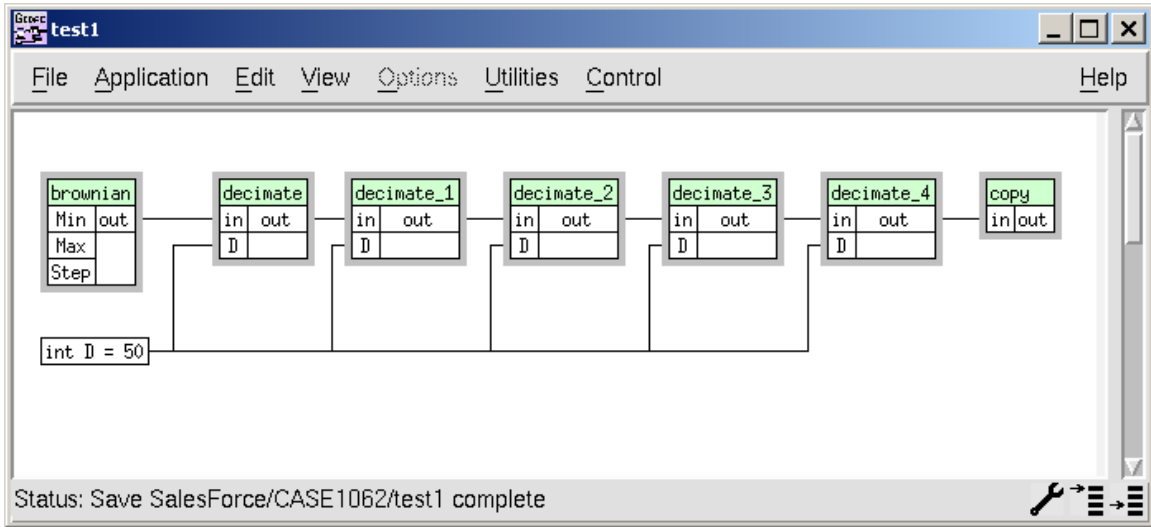
Hierarchical Scheduling

Graphs can be subscheduled hierarchically. That is, a subschedule may itself include a subschedule. For example, if there are boxes in a graph that fire with granularities 1, 10 and 100, then the boxes with granularity 10 and 100 can be made part of the same subschedule. The boxes at a granularity of 100 can be further subscheduled. This multilevel or **hierarchical scheduling** can be specified in the Partition Table using names that include periods to indicate multiple subschedule levels. So for example, the boxes at a granularity of 10 can be in a subschedule labeled "ss" and the ones at 100 can be placed in a hierarchical subschedule labeled "ss.1". The boxes marked ss.1 are placed in a subschedule "1" that is within the subschedule ss. Many levels of subscheduling can be created by adding more periods to the name. The next example illustrates how hierarchical subscheduling can be used effectively to reduce memory usage in a typical sensor application.

Subscheduling to Reduce Memory Usage

Subscheduling can be used to save on overall memory usage. One example of this savings is from sensor processing in which the data being processed is decimated in several stages. The graph below provides a simple illustration of multilayered

decimation and shows how to effectively use hierarchical subscheduling. In the graph, the data rate is decimated five times – each time at a rate of 50:1.



If the primitives are allowed to go to their default granularity, then one can see that the brownian box must fire 312,500,000 times to provide enough data for the downstream copy box to fire once.

Name	Gran Mult	Nat Gran	Max Gran	Granularity	Priority
Schedule 1	1				
brownian		312500000	312500000		0
decimate		6250000	6250000		0
decimate_1		125000	125000		0
decimate_2		2500	2500		0
decimate_3		50	50		0
decimate_4		1	1		0
copy		1	1		0

The static schedule for this graph requires about 1.2 Gbytes of data – far in excess of the memory resources of most target processors.

Name	Size	Priority	Policy	Period
Part default	1275000000			

In order to reduce the memory requirements, the graph can be hierarchically subscheduled with the boxes towards the beginning of the graph at the deepest level of the hierarchy. A four level subscheduling of this graph is illustrated below:

Name	Part	SubSched
brownian	default	1,1,1,1 *
decimate	default	1,1,1,1 *
decimate_1	default	1,1,1 *
decimate_2	default	1,1 *
decimate_3	default	1 *
decimate_4	default	
copy	default	

which creates a granularity table as follows:

Name	Gran Mult	Nat Gran	Max Gran	Granularity	Priority
Schedule 1	1				
SubSched 1	1	50	50		0
SubSched 1.1	1	50	50		0
SubSched 1.1.1	1	50	50		0
SubSched 1.1.1.1	1	50	50		0
brownian		50	50		0
decimate		1	1		0
subenq_1		1	1		0
decimate_1		1	1		0
subenq_2		1	1		0
decimate_2		1	1		0
subenq_3		1	1		0
decimate_3		1	1		0
subenq_4		1	1		0
copy		1	1		0
decimate_4		1	1		0

The above table shows the hierarchy of the subscheduling. For example, the top-level schedule consists of the boxes SubSched1, copy and decimate_4. The SubSched1 box fires at a granularity of 50 and the other two boxes fire at a granularity of 1. The

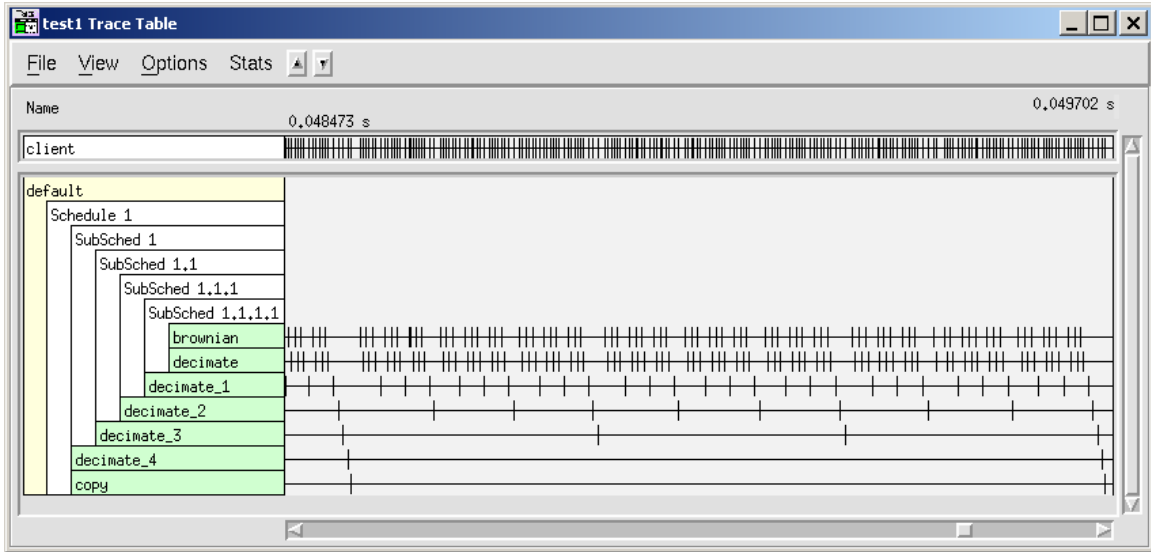
Subched1 schedule consists of the boxes SubSched1.1, the decimate_3 box and the subenq box, which is the interface between SubSched1 and its parent schedule. This breakdown continues for four levels.

By default all of the subschedules have a granularity multiplier of 1, but this can be raised to increase the granularity of the boxes firing within the subschedule. As noted before, the ability to set the subschedule granularity multipliers allows the user to trade off schedule size and granularity. Higher granularity tends to reduce the overhead of the boxes but increases the amount of memory needed by the schedule.

For the above subscheduling, the schedule table looks like this:

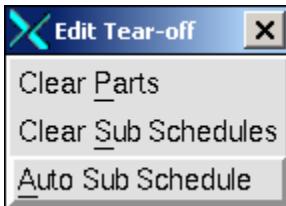
Name	Size	Prior
Part default		
Schedule 1	208	
SubSched 1		
Segment default	200	
Segment parent_memory	8	
SubSched 1.1		
Segment default	200	
Segment parent_memory	8	
SubSched 1.1.1		
Segment default	200	
Segment parent_memory	8	
SubSched 1.1.1.1		
Segment default	200	
Segment parent_memory	8	

Now instead of requiring 1.2 Gbytes of storage for one schedule, only about 200 bytes of storage are required by each of five different schedules. Below is a Trace Table resulting from such a subscheduling. The decimation rate was changed from 50 to 3 so that individual primitive executions at each level of subscheduling are more easily viewed.



Automated Subschedule Generation

For larger graphs, setting subschedules using the Partition Table can be tedious and error prone. Using this table, it is difficult to determine which boxes are in connected sets and also share common granularity factors. To aid the user in setting subschedule boundaries, an automated subscheduling tool is provided. The tool is invoked from the Partition Table dialog menu Edit->Auto Subschedule.



The dialog used to control the autosubscheduling is called the Gain Hierarchy Table and is shown below.

Name	TotalG	Bytes	Div	TDiv	G	Bytes*G	Boxes	Subsched
Schedule 1	1	204	1	1	1	204	2	
1	50	204	1	1	50	10200	1	
2	2500	204	10	10	250	51000	1	1
3	125000	204	50	500	250	51000	1	1.1
4	6250000	204	50	25000	250	51000	1	1.1.1
5	312500000	4	1	25000	12500	50000	1	1.1.1.1

The Gain Hierarchy Table breaks the graph into a hierarchy of connected sets that share common granularity factors. In the above graph, Set 5 is a connected set of primitives (in this case 1) that all run at a natural granularity of 312,500,000. Set 4 contains set 5 and consists of boxes that have a granularity factor of 6,250,000. There are two boxes in set 4. Set 1 consists of a connected set of five boxes that all share a granularity factor of 50.

Similar to the Partition Table, the Gain Hierarchy Table is used to create hierarchical subschedules; however, with the Gain Hierarchy Table the process is automated. The user does control the process. The table already provides the information about which boxes have common granularity factors and which boxes are connected. The following is an explanation of the fields in this dialog:

- Name: Can be the name of a schedule, or an integer label of a set of boxes with the same granularity (known as a granularity-set)
- TotalG: The total granularity of the boxes in this granularity set. This value is the number of times the boxes must fire to complete one cycle
- Bytes: A rough estimate of the number of bytes needed by the boxes in this set if executed at a granularity of 1. Without subscheduling, the boxes in a granularity set are estimated to require $TotalG * Bytes$ bytes of memory
- Div: Divider - tells how many times subschedule will run in a higher-level schedule. $Div = 1$ indicates no subscheduling at this level. A Div other than 1 adds a level of subscheduling for the granularity set and its children sets.
- TDiv: Total divider - is the product of all the dividers from the level granularity-set to the current set.
- G: Granularity of boxes in this set within the subschedule. $G = TotalG / TDiv$
- Bytes*G: An estimate of the number of bytes required by boxes in this granularity set. Typically, the goal of using this table is to create subschedules that control the size of this value.
- Boxes: The number of boxes in this granularity
- SubSched: The name of the subschedule the boxes in this set belong to

The goal is to set the Div field so that the Bytes*G fields can be contained in memory. Only the Div field can be directly modified. Modifying the Div field also modifies the fields with red text. Subschedules are created at any level where the gain divider is not 1. The product of the gain divider and all the higher-level gain dividers given in field TDiv cannot exceed the TotalG column. If the gain divider is other than 1 but TDiv is less than the TotalG column, then the granularity multiplier of the subschedule will automatically be set to $(TotalG / TDiv)$. Thus, the autosubscheduling tool allows the user to control which primitives are members of subschedules and to control the granularity multiplier of these subschedules by setting the gain divider.

The Div field can be set to change the Bytes*G field in one of three ways:

- Edit->Reset: All Dividers: sets all dividers to 1 eliminating subscheduling
- Edit->Set Limit: Dividers are set to attempt to keep Bytes*G less than the Limit entered

- Options->Set Divider: Sets the divider of the selected granularity sets to the value entered

The entries in the Gain Hier Table can be expanded to view the primitives contained within each gain level. Double clicking on any of the levels will expand or collapse the level so the user can see or hide the primitives within that level. Selecting View->Show All Boxes will expand all the levels.

Name	TotalG	Bytes	Div	TDiv	G	Bytes*G	Boxes	Subsched
Schedule 1	1	204	1	1	1	204	2	
copy								
decimate_4								
1	50	204	1	1	50	10200	1	
decimate_3								
2	2500	204	10	10	250	51000	1	1
decimate_2								
3	125000	204	50	500	250	51000	1	1,1
decimate_1								
4	6250000	204	50	25000	250	51000	1	1,1,1
decimate								
5	312500000	4	1	25000	12500	50000	1	1,1,1
brownian								

When the Gain Hier Table is first brought up the gain dividers are set by setting the Bytes*G limit to 10000. In the table above, dividers were set by selecting Edit->Set Limits and entering 100000 into the dialog.

Input Dialog

Enter a schedule byte count limit to automatically set subschedules to keep schedules within that limit

100000

OK Cancel

This setting gives schedule sizes as shown below.

The screenshot shows a window titled "Group 1 Schedule Parameters" with a menu bar (File, Edit, View, Options, Help). Below the menu is a table with columns: Name, Size, Priority, and P. The table contains a hierarchical structure of schedule parameters:

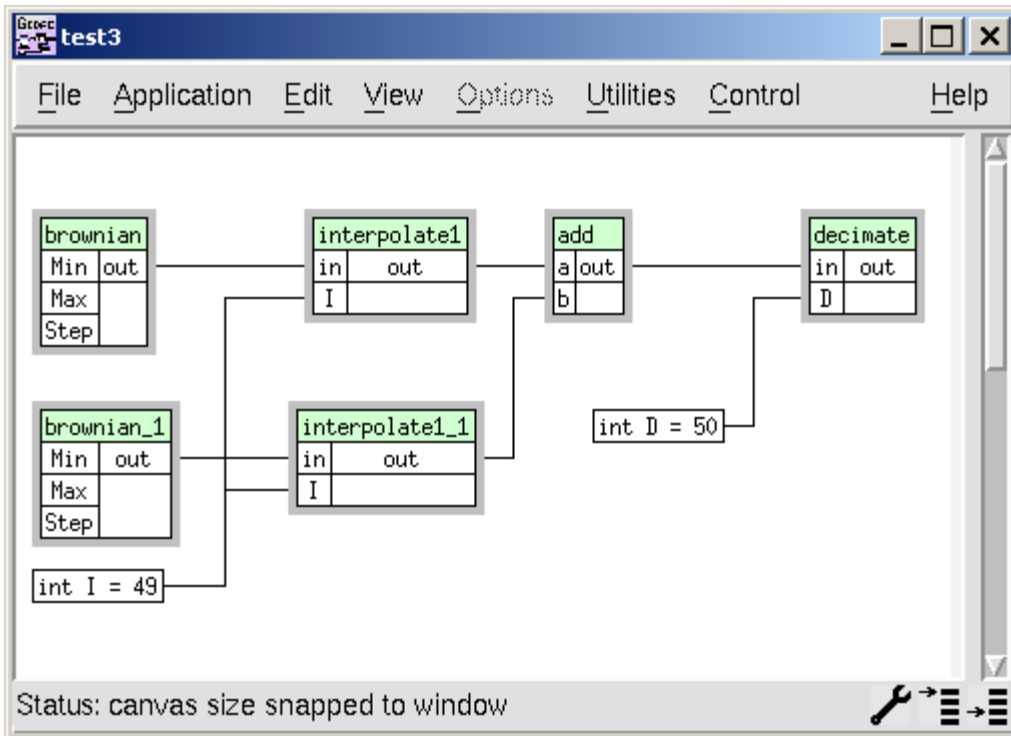
Name	Size	Priority	P
Part default			
Schedule 1	10200	0	
SubSched 1			
Segment default	50000		
Segment parent_memory	1000		
SubSched 1.1			
Segment default	50000		
Segment parent_memory	1000		
SubSched 1.1.1			
Segment default	50000		
Segment parent_memory	1000		

These schedules have a larger size than the minimum 200 bytes, but they are still well within the amount of available memory for a typical target processor. By setting the limit to 100,000, a balance of two conflicting goals has been achieved. The application fits into available memory and the granularity of primitive execution is high enough to maintain efficiency.

Two notes about autosubscheduling; first, the estimate of schedule size in the Bytes field is quite crude and may not be a good estimate of the actual schedule size. Generally the Bytes field is an overestimate; therefore, it may be necessary for the user to set the Div field lower to achieve the desired schedule sizes (as viewable in the Schedule Info dialog). Second, the subschedules created by the autosubscheduling dialog are saved when Group Settings are saved exactly as if the user had set the subschedules and granularity multipliers without using the dialog. For example, if the user set the Limit to 100000 and then changed the graph, the subschedules would not be automatically generated for any boxes added unless the user brought up the autosubschedule tool and reset the Limit to 100000 again.

Multiple Parents

Subschedules must form a strict hierarchy. That is, a subschedule cannot belong to two different parent subschedules. Occasionally the gains in a graph do not form a strict hierarchy. In such a case, the user may choose among the different ways of forming the hierarchy. A simplified example of how this can happen is seen in the following graph:

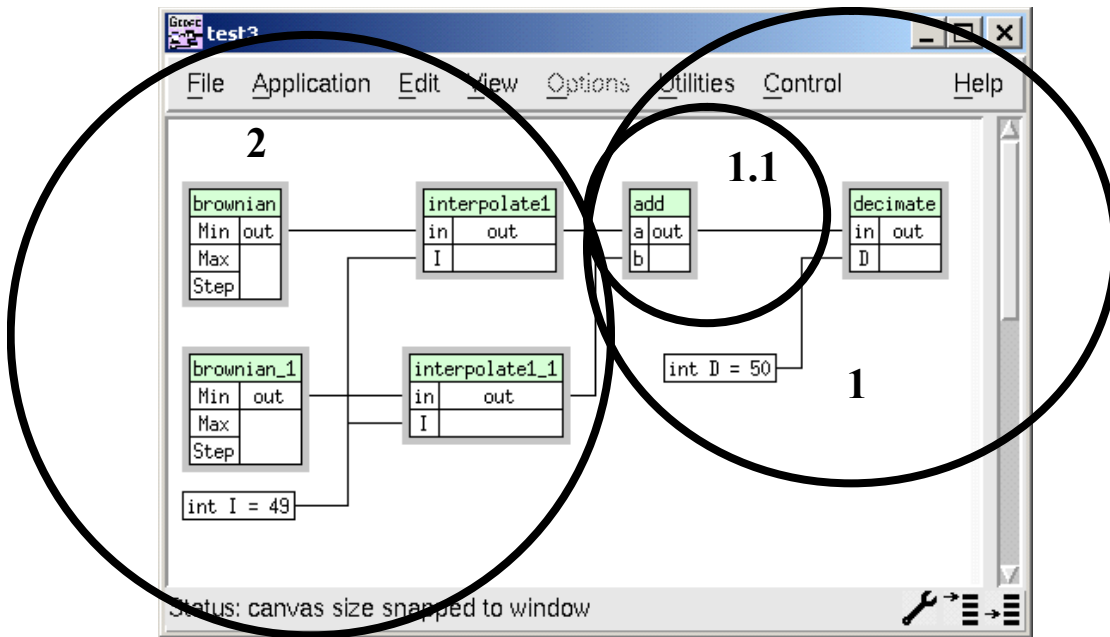


In the above graph, the interpolate boxes produces 49 tokens on their output for each token on its input. The decimate box requires 50 tokens on its input to produce one token on its output. For the schedule to complete a cycle of the data flow, the brownian and interpolate boxes must run 50 times, the decimate box must run 49 times and the add box must run $49 \times 50 = 2450$ times. These numbers are summarized in the Gain Hierarchy Table seen below:

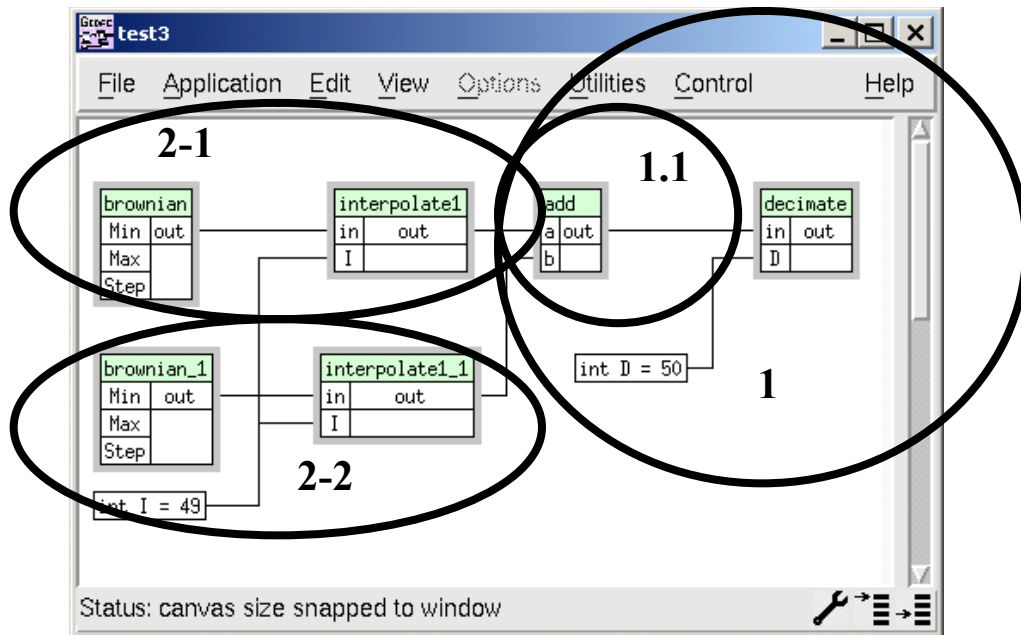
Name	TotalG	Bytes	Div	TDiv	G	Bytes*G	Boxes	Subsched
Schedule 1	1	0	1	1	1	0	0	
1	49	204	49	49	1	204	1	1
decimate								
2	2450	8	50	2450	1	8	1	1.1
add								
3	50	400	50	50	1	400	4	2
brownian								
brownian_1								
interpolate1								
interpolate1_1								
2								
add								

Set 2, which contains a single add box with a granularity of 49×50 , can either be placed in a group parented by Set 1 with a granularity of 49 or by Set 3 with a granularity of 50. A choice must be made as to whether Set 2 is a child of Set 1 or Set 3. In the above table,

Set 2 is parented by Set 1. That Set 2 is not a child of Set 3 is indicated by the Set 2 line under Set 3, which is white in color and has no information in the different text fields.



Because the add box is not part of Set 3, Subschedule 2 is actually now illegally disconnected. To avoid this problem, the subschedule is automatically broken into two separate subschedules labeled 2-1 and 2-2 as follows:



This structure can be seen in the Schedule Info table:

Group 1 Schedule Parameters

File Edit View Options Help

Name	Size	Priority	Policy	Period	Retry	Memory Type	Ex Membership
Part default							
Schedule 1	19600	0	dataflow			default	shared
SubSched 1							shared
Segment default	8					default	shared
Segment parent_memory	400					parent_memory	shared
SubSched 1.1	16					parent_memory	shared
SubSched 2-1							shared
Segment default	8					default	shared
Segment parent_memory	200					parent_memory	shared
SubSched 2-2							shared
Segment default	8					default	shared
Segment parent_memory	200					parent_memory	shared

Group 1 Gain Hier Table

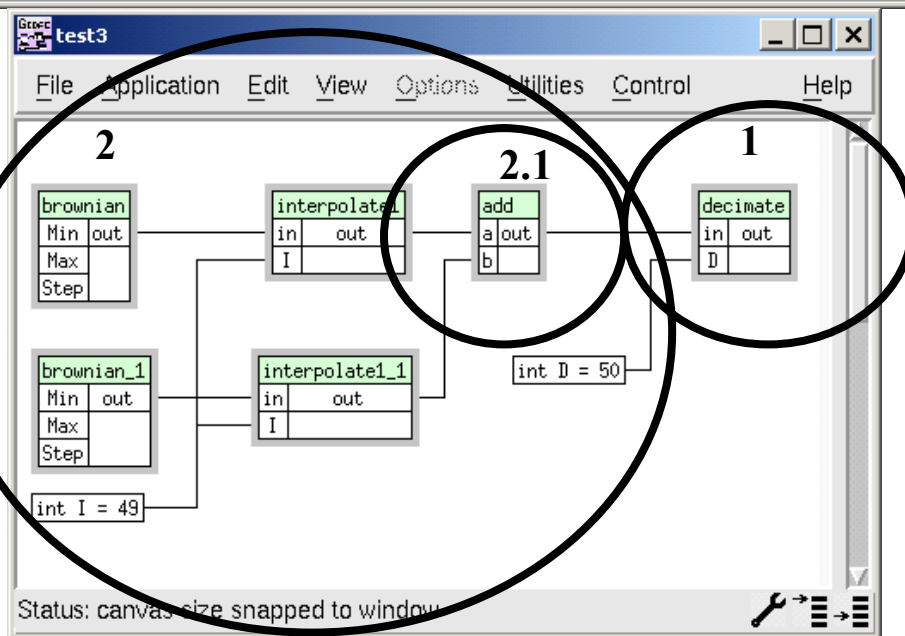
File Edit View Options Help

Name	TotalG	Bytes	Div	TDiv	G	Bytes*G	Boxes	Subsched
Schedule 1	1	0	1	1	1	0	0	
1	49	204	49	49	1	204	1	1
decimate								
2	2450	8	50	2450	1	8	1	1,1
add								
3	50	400	50	50	1	400	4	2
brownian								
brownian_1								
interpolate1								
interpolate1_1								
2								
add								

- Options Tea... X
- Expand
 - Collapse
 - Hide
 - Show All Boxes
 - Set Divider...
 - Set Branch...

Group 1 Gain Hier Table

Name	TotalG	Bytes	Div	TDiv	G	Bytes*G	Boxes	Subsched
Schedule 1	1	0	1	1	1	0	0	
1	49	204	49	49	1	204	1	1
decimate								
2								
add								
3	50	400	50	50	1	400	4	2
brownian								
brownian_1								
interpolate1								
interpolate1_1								
2	2450	8	49	2450	1	8	1	2.1
add								



Group 1 Schedule Parameters

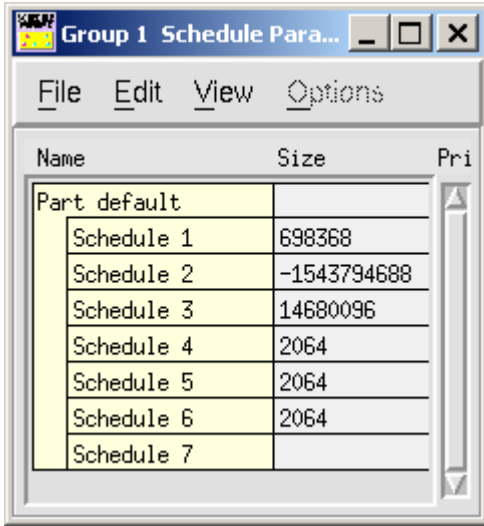
Name	Size	Priority	Policy	Period	Retry	Memory Type	Ex Membership
Part default							
Schedule 1	9800	0	dataflow			default	shared
SubSched 1							shared
Segment default	8					default	shared
Segment parent_memory	200					parent_memory	shared
SubSched 2							shared
Segment default	208					default	shared
Segment parent_memory	200					parent_memory	shared
SubSched 2.1	16					parent memoru	shared

As a side note, the automated subscheduling base on setting the memory limit would not have created a nested subschedule for the add box. A subschedule is avoided here because there is only a single box in the granularity set. A subschedule with just one box

has no benefit. In order to place the add box in its own subschedule, it was necessary to set the Div parameter using the Options->Set Divider menu.

Example of a More Complex Graph

The use of the new tools was demonstrated on a customer's graph. Without the use of the tool, the graph would contain schedules with sizes marked as:



The screenshot shows a window titled "Group 1 Schedule Para..." with a menu bar containing "File", "Edit", "View", and "Options". Below the menu bar is a table with three columns: "Name", "Size", and "Pri". The table contains the following data:

Name	Size	Pri
Part default		
Schedule 1	698368	
Schedule 2	-1543794688	
Schedule 3	14680096	
Schedule 4	2064	
Schedule 5	2064	
Schedule 6	2064	
Schedule 7		

The negative size value indicates that the schedule size actually exceeds the 32 bit address space (size is greater than 4 Gbytes). The actual size of Schedule 2 is 47 Gbytes.

Bringing up the autosubschedule tool for this graph automatically creates subschedules using the default Bytes*G limit of 10000 bytes. The result for the given graph is seen below:

Name	TotalG	Bytes	Div	TDiv	G	Bytes*G	Boxes	Subsched
Schedule 1	1	1752616	1	1	1	1752616	21	
1	128	4	1	1	128	512	1	
2	128	4	1	1	128	512	1	
Schedule 2	1	0	1	1	1	0	0	
1	25	7340032	25	25	1	7340032	3	1
2	12800	7196	64	1600	8	57568	11	1.1
3	25600	5376	1	1600	16	86016	2	1.1
4	51200	896280	32	51200	1	896280	27	1.1.1
5	256000	4	1	51200	5	20	1	1.1.1
6	6553600	4	1	51200	128	512	1	1.1.1
7	13107200	4	1	51200	256	1024	1	1.1.1
8	17203200	4	1	51200	336	1344	1	1.1.1
9	22937600	4	1	51200	448	1792	1	1.1.1
10	5734400	8	1	1600	3584	28672	1	1.1
11	512	1020	8	8	64	65280	10	2
2								
Schedule 3	1	5505024	1	1	1	5505024	16	
1	2	20	1	1	2	40	3	
2	2	20	1	1	2	40	3	
3	2	5505024	2	2	1	5505024	1	3
4	896	16384	112	224	4	65536	5	3.1
5	2	20	1	1	2	40	3	
Schedule 4	1	2052	1	1	1	2052	4	
1	512	4	1	1	512	2048	1	
Schedule 5	1	2052	1	1	1	2052	4	
1	512	4	1	1	512	2048	1	
Schedule 6	1	2052	1	1	1	2052	4	
1	512	4	1	1	512	2048	1	

Schedule 2 is broken into two different top-level subschedules, and one of the subschedules is further broken into three levels. Note the granularity set numbered 11 is an alternative parent to the granularity set 2. In this case, we chose to keep this as the default hierarchy. The resulting schedule table is:

Name	Size	Priority	Policy	Period	Retry	Memory Type
Part default						
Schedule 1	698368	0	dataflow		1 schedules	default
Schedule 2	512000	0	dataflow			default
SubSched 1						
Segment default	9175040					default
Segment parent_memory	20480					parent_memory
SubSched 1.1						
Segment default	415776					default
Segment parent_memory	28992					parent_memory
SubSched 1.1.1						
Segment default	917000					default
Segment parent_memory	3584					parent_memory
SubSched 2-1						
Segment default	256					default
Segment parent_memory	12800					parent_memory
SubSched 2-2						
Segment default	256					default
Segment parent_memory	12800					parent_memory
SubSched 2-3						
Segment default	256					default
Segment parent_memory	12800					parent_memory
SubSched 2-4						
Segment default	256					default
Segment parent_memory	12800					parent_memory
SubSched 2-5						
Segment default	256					default
Segment parent_memory	12800					parent_memory
Schedule 3	5505024	0	dataflow			default
SubSched 3						
Segment default	7340032					default
Segment parent_memory	1835008					parent_memory
SubSched 3.1						
Segment default	32768					default
Segment parent_memory	65536					parent_memory
Schedule 4	2064	0	dataflow			default
Schedule 5	2064	0	dataflow			default
Schedule 6	2064	0	dataflow			default
Schedule 7		0	periodic	0,5 secs		

Note that SubSched 2 is broken up into five different subschedules. As described in the previous section, granularity set 11 has become disconnected because granularity set 2 was not included in it.

The largest memory segment contains 9 million bytes. The memory allocation summary for the graph is now:

```

BLOCK default, size = 29421908:
  PERC,    TOTAL,    BYTES,  BLOCKS:  TYPE[ELEMENTS]
  0.01,    2362,    2362,    49:    char[2362]
  0.01,    2462,    100,    25:    char*[25]
  0.02,    6046,    3584,    1:    complex[448]
  1.94,   569982,   563936,   47:    float[140984]

```

1.94,	569994,	12,	3: float*[3]
1.94,	569998,	4,	1: float**[1]
1.94,	571198,	1200,	292: int[300]
1.94,	571386,	188,	43: int*[47]
1.96,	575646,	4260,	15: DynamicQueue[15]
1.96,	575846,	200,	18: DynamicQueue*[50]
1.96,	576134,	288,	12: DynamicQueueArray[12]
1.96,	576478,	344,	1: DynamicSchedule[1]
1.96,	577342,	864,	17: MemType[27]
1.96,	578098,	756,	10: ParentAddr[21]
1.97,	578146,	48,	1: PositionChange[4]
...			
1.97,	579474,	80,	4: SchedHeap[4]
2.00,	587702,	8228,	17: Schedule[17]
2.00,	587914,	212,	11: Schedule*[53]
2.00,	588114,	200,	10: ScheduleAddrInfo[10]
2.00,	589738,	1624,	17: SizeChange[203]
2.08,	611338,	21600,	34: StaticClosure[216]
2.08,	612922,	1584,	35: StaticClosure*[396]
...			
			: ZERO MEMORY
16.38,	4819674,	4203720,	15: QueueData[4203720]
100.00,	29421874,	24602200,	11: ScheduleData[24602200]

The summary shows how much memory the blocks of various types require. QueueData and ScheduleData are affected by subscheduling and account for 98% of the memory used. The total memory used is now only 29 Mbytes – well within the capability of many target processors and a 2000:1 reduction in memory over the original 47 Gbytes.

Conclusion

Subscheduling can be an effective method for both improving speed by maintaining cache coherency and for reducing memory usage. Subschedules can be directly set from the Partition Table or set in an automated fashion that is controlled by the user with the autosubschedule tool.

The autosubschedule tool simplifies the process of creating subschedules. While the user of the autosubschedule tool has some control over how the subschedules are created – just bringing up the tool is enough to achieve memory improvements. The autosubschedule tool makes it much easier for a user to achieve good results with subscheduling and requires minimal understanding by the user in order to achieve the benefits.