Report of High Level Synthesis retiming task

By Ludwig Meysel and Mitja Stachowiak
Report - 19th of March 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

Introduction

The task was to implement a loop retiming based on a simulated annealing approach. The aim of retiming is to shift nodes in a data flow graph to future or past iterations of the loop, to get hopefully shorter initiation intervals after scheduling.

List scheduling with different resource constrained files is used in this program to evaluate the effect of retiming.

Contents

1 Program structure

1.1 Useage

The program expects 5 parameters:

1. Input file name: This can either be a .dot file or a directory containing multiple of such files.
2. Resource file name: The file holding the resource constraints.
3. Quality: Integer value representing the quality factor of SA.
4. Output direcoty: In this folder will the scheduled results be saved.
5. (optional) The last parameter can be *scheduleAsCost*. In this case, a complete, resource constrained schedule will be done in each SA-cycle to evaluate the real schedule length as the cost function. If this parameter isn't set, the length of the maximum path is used as cost function.

During execution, the program will print a table holding the number of nodes in the graph, the initial cost, the cost after retiming, and the final length of the resulting schedule of each processed .dot-file.

1.2 Classes

The following classes were written or modified for this program:

- **Node**

  This class is part of the framework and each instance represents one operation in the data flow graph. Edges can have a weight, representing the offset of the loop iteration. Each node has a predecessor and a successor-map, which holds an integer representing the edge weight as values and the neighbor nodes as keys. The original version of this class could produce inconsistencies, where one node is successor of an other node, but this other node is not predecessor of the first node. For this program, the methods for linking nodes are modified, such that the graph is always consistent.

  As a new functionality, the depth-value is now auto-updating. As this value is needed very often, it can even become more efficient, to update this value dynamically. Otherwise, it was necessary to reset the depths of all nodes after each retiming operation.

  Each node now has a tmp1-value, which is free to use for other algorithms. This is usually simpler and faster than to store such extra-values in additional hash maps. If an algorithm wants to use tmp1, it has to set the graphs tmp1Used-variable to true. If it already is true, an exception has to be thrown. This should only occur during development, if two nested algorithms use this value.

- **Graph**

  As described above, the Graph-class now has a variable tmp1Used. Because the nodes keep the graph's consistency by them selves, the Graph.link-function was simplified as well.

- **ListScheduler**

  The new class ListScheduler extends Scheduler to enable list scheduling on graphs. The graph can contain weighted loop edges, which will safely be ignored by the list scheduler. Before starting the scheduling, the constraints have to be set to the constraints-value.

  The list scheduler uses the length of the longest path beyond each node as the priority criterion.

- **Retimer**

  The Retimer-class, is the basis for retiming algorithms. It provides methods for finding the longest directly connected (zero edge weight) path in the graph.

- **SAretimer**

  This class extends the basic retimer and implements the simulated annealing. The algorithm usually uses the longest path in the graph as cost function, but can also use the real schedule length by giving a scheduler to the SAretimer.scheduler-value.

  The retiming returns the start cost, the end cost and the number of cycles, processed for the algorithm.

2 Algorithm improvements

2.1 Search for rotatable nodes

The principal simulated annealing algorithm is well known. But for the given problem, some implementation details have to be solved.

Retiming means moving particular nodes or sub-graphs to different iterations. To construct a set of all rotatable sub-graphs would be an unsolvable problem and even to find such sub-graphs in a structured way is very difficult. So for this program, only single nodes were rotated, assuming, that the rotation of each possible sub-graph could be replaced by a sequence of such atomic rotations.

When searching rotatable nodes, it is possible, to search for rotations to future loop iterations or rotations to past loop iterations. It is possible to choose the direction of rotation randomly for each rotation. But like in the salsa-algorithm, this would lead to long run-times to get a good result. It happens too often, that for example a node, which was rotated into future gets rotated back into past, before any related node, which can only be rotated into future, after this node was rotated into future, can be processed.

To get the algorithm rotating larger sub-graphs, two improvements were done:

- When searching for rotatable nodes, no random selection is done, but all nodes are stored in an array, which gets mixed. Then, each node is tested for rotatability, but the array gets only re-mixed, after each node was regarded.
- The direction of rotation gets not changed for each rotation but there is a factor *dirChangeInterval*, which specifies after how many iterations at maximum through the mixed array, the direction should change. The exact number of iterations is chosen randomly and the new direction is chosen randomly, too.

It can occur, that no node in the graph can be rotated to future or past. In this case, the direction is changed immediately. If still no node can be rotated, the algorithm stops.

This two improvements caused the algorithm to yield much better results for the same number of cycles.

2.2 Initial temperature

Common implementations of simulated annealing run some random changes, measure the deviation in the result and set the initial temperature to a twentyfold of this deviation. It seems, that for the retiming-problem it is sufficient, just to set the temperature to a value, such that a double of the cost will be accepted with 50% probability.

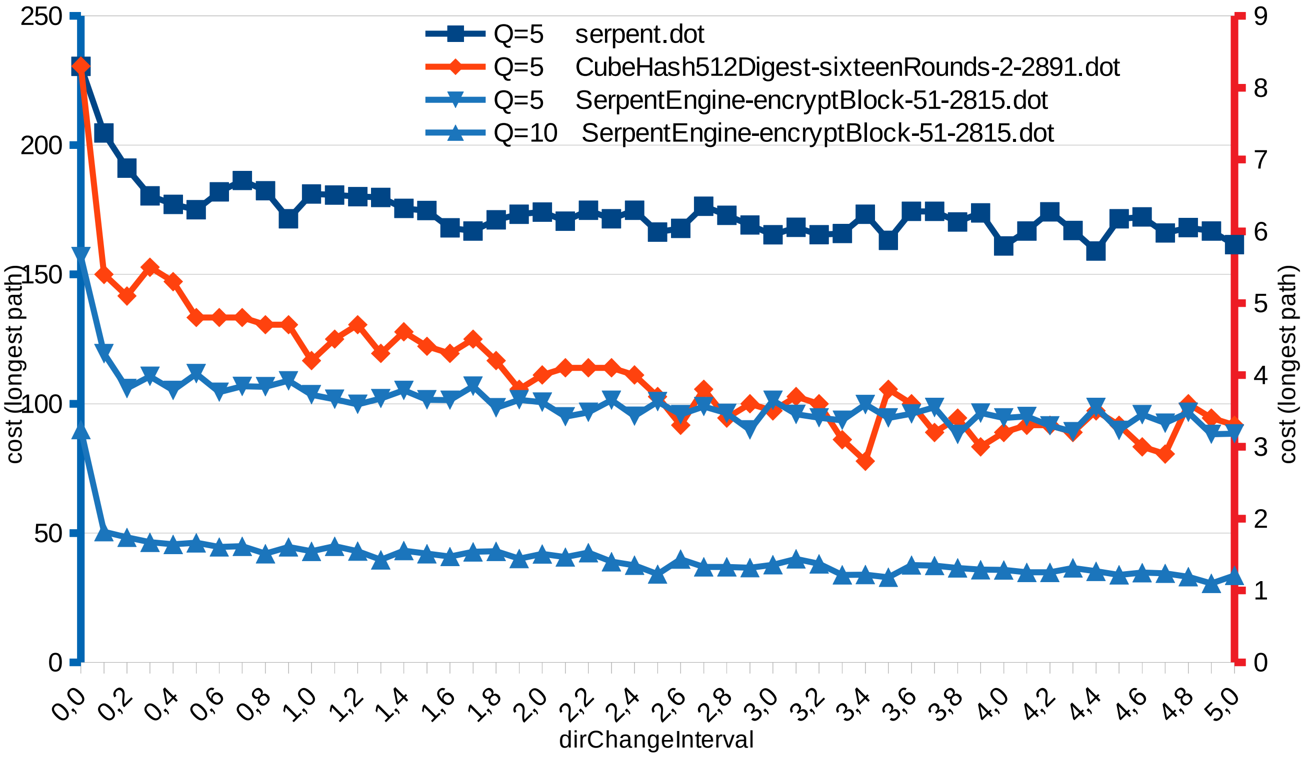
This is because the range of cost change is comparatively small for this type of problem.

3 Results and fine-tuning

First of all, the dirChangeInterval was set to 2 and all available test graphs were retimed with several quality values. It turned out, that a lot of graphs didn't show any improvement or quickly converge to a fixed optimum even with small quality factors, which didn't change when increasing the quality. Just a few graphs became better and better with increasing quality. Very interesting is the serpent.dot-graph. Retiming this graph with extremely high qualities shows, that the longest path can become less than 4, while the initial length is 315.

3.1 Optimization of dirChangeInterval

Next, it is questionable, what is a good value for the dirChangeInterval. To answer this question, three graphs were retimed with different values for the dirChangeInterval. The retiming was repeated ten times for each value of dirChangeInterval and the average was taken. Because a non-deterministic random procedure is used in this program, the results of each run of the algorithm can change.



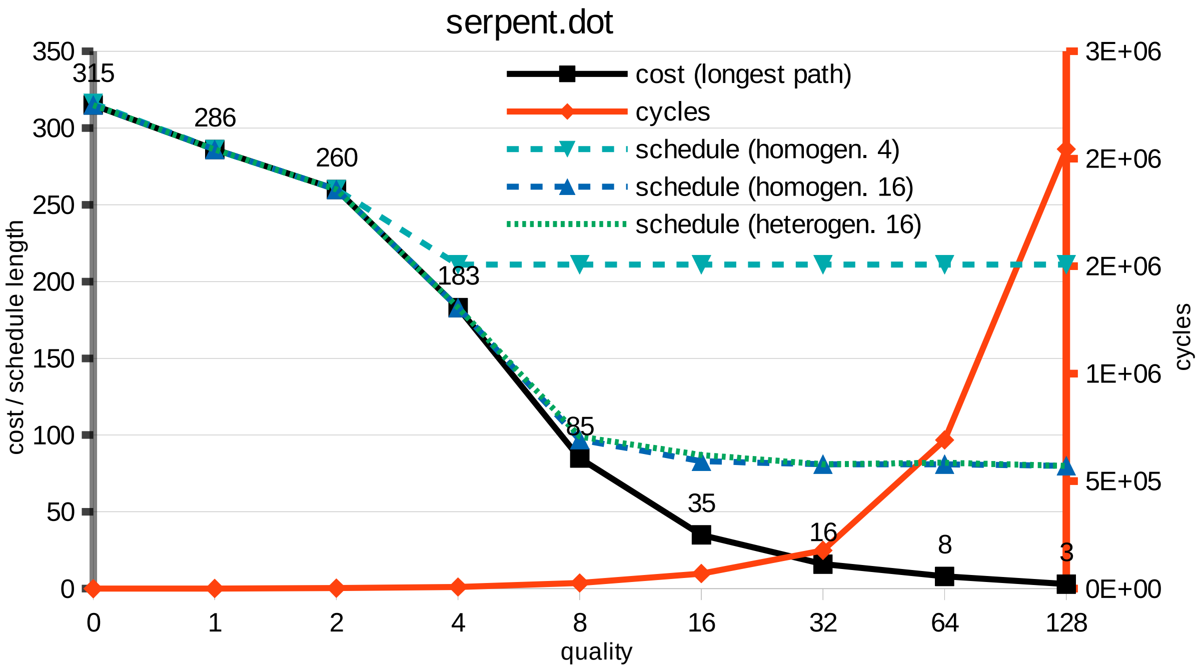
It can be seen, that larger values for dirChangeInterval tend to better results. Remember: This value has no influence on the runtime. So there is no reason, not to take a value like 5 for this interval.

This value of 5 means, that in average all 2.5 search cycles through the randomly sorted array, the direction is reset to a random value (future or past). This chart also shows, that the whole strategy for picking rotatable nodes might be questionable. It should be tried to remember, weather a node was rotated in last cycle and weather is was rotatable. Nodes which have been rotated a short time ago, should not be rotated back too early. But nodes, which haven't been rotatable in previous cycles but are now, should absolutely be rotated.

3.2 Influence of quality factor

The quality factor determines, how often the randomly sorted array is completely searched and re-mixed for each temperature.

The retiming usually optimizes the longest path in the graph, which is somehow correlated with the final schedule length. To get an impression, which qualities lead to which cost (path length) and schedule length, the complex serpent.dot is retimed for different qualities and scheduled for different resource constraints:



It can be seen, that this graph can get an immense benefit from retiming, but according to the number of resources in the constraints file, the final schedule length is saturating at a certain point. Before this point, the schedule length is highly correlated to the longest path in the graph.

Because no available resource constraint file has more than 16 processing elements, it seems, that quality factors larger than 10 are not necessary.

The number of cycles, plotted in this graph is the number of repetitions of the inner loop of the SA-algorithm.

The same chart for the SerpentEngine-encryptBlock-51-2815.dot looks quite similar. For the CubeHash512Digest-sixteenRounds-2-2891.dot, the cost function goes down from 30 to 2 at a quality of 16 but the schedule lengths already saturate at a quality of 2. This early saturation occurs for many graphs.

3.3 General overview table

The following table shows the cost (longest path) and schedule length of all available graphs for no retiming, quality 5, quality 10 and quality 10 with the schedule length as the cost function.

file name	nodes	No retiming Quality = 0		Quality = 5			Quality = 10			scheduleAsCost Quality = 10	
	▼	cost	sched	cost	sched	cycles	cost	sched	cycles	sched	cycles
SHA256Digest-processBlock-735-750.dot	3	2	2	2	2	111	2	2	274	2	215
MD5Digest-processBlock-1896-1911.dot	3	2	2	2	2	94	2	2	216	2	186
ECOH256Digest-engineReset-37-53.dot	4	4	4	2	2	124	2	2	324	2	326
ADPCMn-decode-771-791.dot	5	5	5	2	2	301	2	2	446	2	640
SerpentEngine-makeWorkingKey-150-172.dot	5	5	5	2	2	216	2	2	533	2	652
testCyclic.dot	7	10	10	4	4	308	4	4	710	4	733
test.dot	7	10	10	4	4	402	4	4	809	4	821
ADPCMn-decode-524-553.dot	7	8	8	4	4	308	4	4	1220	4	604
ADPCMn-decode-803-832.dot	8	9	9	4	4	544	4	4	979	4	931
FFT-#init#-14-53.dot	9	8	8	2	2	551	2	2	1337	2	1234
ADPCMn-decode-559-599.dot	9	5	5	2	2	491	2	2	1160	2	801
IDCT-getTransformedInt1DFast-31-65.dot	10	11	11	4	4	569	4	4	1489	4	1475
IDCT-getTransformedInt1DFast-134-169.dot	10	11	11	4	4	685	4	4	1475	4	1271
SerpentEngine-makeWorkingKey-150-195.dot	10	6	6	2	2	835	2	2	1120	2	1584
WhirlpoolDigest-processBlock-2-37.dot	11	7	7	2	4	766	2	4	1197	4	980
lectureLIST.dot	11	10	10	4	4	872	4	4	1529	4	976
ADPCMn-decode-425-472.dot	12	8	8	4	4	591	4	4	1182	4	1503
WhirlpoolDigest-increment-10-53.dot	12	9	9	2	2	802	2	2	1520	2	1533
IDCT.dot	13	15	15	4	4	1084	4	5	2398	4	2717
SHA1Digest-processBlock-17-67.dot	17	10	10	2	4	1254	2	4	3112	4	1269
lectureExample.dot	17	19	19	14	14	705	14	14	1370	14	1267
SIMD512Digest-compress-1566-1661.dot	19	12	12	2	4	1035	2	4	3012	4	1687
SerpentEngine-makeWorkingKey-179-250.dot	20	12	12	2	4	1235	2	4	3380	4	1599
SerpentEngine-makeWorkingKey-15-82.dot	21	9	9	2	4	1604	2	4	3643	4	1931
BLAKE256Digest-processBlock-160-230.dot	21	13	13	4	4	1068	4	4	2560	4	3041
GrayscaleFilter-filter-13-113.dot	21	16	16	4	4	1284	4	4	3020	4	3094
SerpentEngine-makeWorkingKey-86-155.dot	21	12	12	2	4	1642	2	4	3454	4	2176
ECOH256Digest-compress-49-121.dot	23	10	10	2	4	1258	2	4	3014	4	1675
ADPCMn-decode-631-729.dot	23	5	5	2	2	1244	2	2	2945	2	2555
AESrkgcyclic.dot	24	13	13	4	4	1893	4	4	3280	4	4288
XTEAEngine-setKey-70-169.dot	27	9	9	2	6	1954	2	6	4916	6	3308
ADPCMn-decode-271-381.dot	27	7	7	2	4	2325	2	4	4444	4	2351
SobelFilter_inline-sobelEdgeDetection-79-203.dot	27	44	44	18	18	2124	18	18	4055	18	3452
XTEAEngine-encryptBlock-21-142.dot	34	21	21	20	20	906	20	20	1964	20	1899
WhirlpoolDigest-#init#-310-427.dot	36	18	18	4	6	2648	4	6	7037	6	4926
SHA256Digest-processBlock-17-148.dot	36	11	11	2	4	2587	2	4	5164	4	3662
SerpentEngine-makeWorkingKey-179-322.dot	40	13	14	2	6	3607	2	6	7516	6	3528
XTEAEngine-setKey-4-123.dot	41	10	11	2	6	3585	2	6	7320	6	3312
SerpentEngine-makeWorkingKey-15-150.dot	42	10	11	2	6	3598	2	6	8246	6	3031
PETrigonometry-cordic-88-287.dot	42	36	36	36	36	2521	36	36	4670	36	4528
SerpentEngine-makeWorkingKey-86-225.dot	42	13	14	2	6	3694	2	6	8970	6	3725
TwofishEngine-RS_MDS_Encode-5-196.dot	44	22	22	2	3	4063	2	3	7945	3	5921
ContrastFilter-filter-13-252.dot	47	39	57	18	36	3614	18	36	11665	36	7970
FFT-fft3-688-929.dot	50	12	16	4	14	4250	4	14	8958	12	7640
FIR-main-75-264.dot	53	11	19	4	18	3508	4	18	8750	18	5173
SkipjackEngine-init-45-238.dot	54	12	16	4	12	4844	4	13	10714	12	10644
SkipjackEngine-encryptBlock-150-445.dot	59	19	19	2	8	4686	2	8	12437	8	4504
SkipjackEngine-encryptBlock-97-392.dot	59	35	35	2	8	4543	2	8	11743	8	4193
ECOH256Digest-mixColumn-3-238.dot	64	18	18	4	6	5417	4	6	12775	6	12492
WhirlpoolDigest-processBlock-50-321.dot	76	19	19	2	14	5991	2	14	13382	14	13205
ECOH256Digest-compress-839-1065.dot	76	21	24	4	12	5815	4	12	13673	10	13781
WhirlpoolDigest-processBlock-359-635.dot	78	19	19	2	16	7469	2	16	17724	16	6069
SHA1Digest-processBlock-332-693.dot	82	21	21	5	6	5856	5	6	13304	6	9851
SwizzleFilter-filterImage-13-423.dot	82	30	31	4	18	7848	4	18	17749	16	19586
SIMD512Digest-fft64-1537-1965.dot	91	14	19	4	16	8515	4	16	15700	14	17847
SHA1Digest-processBlock-105-486.dot	92	21	21	5	7	5102	5	7	11230	7	9492
SHA1Digest-processBlock-559-950.dot	97	21	21	5	7	5902	5	7	15638	7	7606
FIR-main-75-454.dot	106	11	35	4	38	8440	4	38	17203	36	25729
SIMD512Digest-compress-63-471.dot	106	13	19	4	16	9471	4	16	19467	14	16911
FFT-fft3-387-948.dot	111	19	31	18	32	4342	18	32	8747	30	26767
SIMD512Digest-compress-1878-2280.dot	111	16	24	4	18	9254	4	18	22362	16	22228
SIMD512Digest-fft64-3640-4116.dot	122	14	20	4	16	10252	4	16	18808	14	19430
ECOH256Digest-AES2RoundsAll-2-666.dot	179	21	36	2	34	15616	2	34	35300	34	19620
SIMD512Digest-compress-1682-2248.dot	187	15	31	4	28	16238	4	28	36051	28	22980
TwofishEngine-#init#-3600-4541.dot	204	12	30	2	28	20389	2	28	37570	28	54412
WhirlpoolDigest-#init#-75-755.dot	204	14	27	2	19	17193	2	19	35282	18	53065
TwofishEngine-encryptBlock-90-1017.dot	250	76	76	41	45	6777	41	43	14160	42	14891
SerpentEngine-makeWorkingKey-245-1724.dot	285	23	51	2	48	26476	2	48	59562	48	64416
BLAKE256Digest-processBlock-189-1577.dot	308	43	47	36	36	9779	36	36	24556	36	34068
RadioGatun32Digest-processBlock-170-1524.dot	309	17	29	8	34	13177	8	33	35424	31	42366
TwofishEngine-setKey-262-1953.dot	331	14	68	4	72	26450	4	72	65819	68	85591
TwofishEngine-setKey-113-1859.dot	333	19	68	18	68	12801	18	68	27593	68	30609
SHA256Digest-processBlock-119-1722.dot	378	57	57	31	31	9744	31	32	23477	31	22490
CubeHash512Digest-sixteenRounds-2-2891.dot	546	30	119	3	112	44471	3	112	114275	112	137359
SerpentEngine-encryptBlock-51-2815.dot	563	225	225	99	99	20586	21	54	61837	-	-
serpent.dot	682	315	315	150	150	13733	61	91	32566	-	-

4 Conclusion

For future improvements, a better node selection strategy, which tends to shifting of larger sub-graphs could be found, as described in the ending of chapter 3.1.

From the general overview table, it could be seen, that scheduling different graphs having the same cost (longest path), can lead to different schedule lengths, differing in one or just a few cycles. While using the final schedule length as the cost function has only a very small benefit, but increases the runtime dramatically, the longest path approach should be kept, but to get rid of the small deviations in the schedule lengths, it maybe makes sense to use the schedule length as the cost function only in the last temperature-step of SA.

One problem, not mentioned yet is, that retiming of nodes can cause a need for additional variables, which store the result of nodes for future schedules. So excessive retiming beyond the saturation of the resource constrained schedule could have negative effects on the total result quality.

In future projects, a solution should be investigated, that can detect the saturation and changes the cost function then to minimize the number of required variables while keeping the longest path constant.