Mini-task report: SDC with simulated annealing

Ludwig Meysel, Mitja Stachowiak

TECHNISCHE
UNIVERSITÄT
DARMSTADT

1 Introduction

The task was to implement a simulated annealing approach (SA) for SDC (system of difference constraints). LPsolve is used to get a schedule for a given set of constraints. The SA-algorithm mutates the order of the constraints to reduce the number of clock cycles of the schedule.

2 Resources

The constraints consist of data flow - and resource constraints. The data flow constraints determine, that no operation must start, before all predecessors have obtained a result. The resource constraints prevent, that the same resource is used twice at the same time.[1]
Each hardware has a certain amount of resources and resource types. There is a fixed list of operations given in the framework:

Operation name (ar)	delay	weight
MEM	2	9.0
ADD	1	1.0
SUB	1	1.4
MUL	4	2.3
DIV	18	4.3
SH	1	2.0
AND	1	2.0
OR	1	2.0
CMP	1	2.1
OTHER	1	1.0
SLACK	1	0.0

Each resource type can support multiple operations. For this project, the resource(types) are assumed to be overlap-free:

$$\neg\exists R_1, R_2 \in Resources; Op_1, Op_2 \in Operations : Op_1 \in R_1 \wedge Op_1 \in R_2 \wedge Op_2 \in R_1 \wedge Op_2 \notin R_2$$

Each resource can handle one operation within a certain time (delay). Multiple resources of the same type may exist.

3 Simulated Annealing

The principal structure of any simulated annealing looks like this:

```
S = RandomConfiguration();
T = InitialTemperature();
while (ExitCriterion()==false) {
  while (InnerLoopCriterion() == false) {
    S_new = Generate(S);
    Δ C = Cost(S_new)-Cost(S);
    r = random(0,1);
    if (r < e^(-ΔC/T)) S = S_new
  }
  T = updateTemperature();
}
```

The implementation is located in scheduler/SASDC.java:schedule. The parameters are:

- *Random Configuration* ...
- *Initial Temperature* is determined by applying n(nodes) random changes and saving the costs of each change. T is then $20 * standardDeviation(costs)$.
- *Exit Criterion* is the condition, when the simulated annealing should stop. For each temperature, the number of applied changes and the number of accepted changes is counted. When less than 12% of the changes are accepted, the algorithm stops.
- *Update Temperature* decreases T by a factor tu, which depends on the acceptance ratio as well:

acceptance ratio (ar)	temperature factor (tu)
> 96%	0.5
96 .. 80%	0.9
80 .. 15%	0.95
< 15%	0.8

- *Inner Loop Criterion* determines, how many changes are tested for the same temperature. Each change usually moves one node in the ordering of constraint-equations. The larger the number of nodes becomes, the more often each node should be moved, so the number of iterations should depend on the node count. Further more, there is a quality factor $\in [1..10]$ for the algorithm, which can be passed via the third program argument. The formula $n_{inner} = \left\lceil quality * n_{nodes}^{4/3} \right\rceil$ is known to yield a result, thats quality belongs to the given quality.

4 Details on implementation

The SDC scheduler with simulated annealing is implemented in the `SASDC`-class. The constructor takes the resource constraints and the scheduling quality as parameters. The quality is forwarded to the inner-loop-criterion, which is explained in chapter 3.

4.1 The node list

The actual implementation of the SA algorithm is straight forward, therefore the real point of interest is how the configuration for a schedule is created and how it can be modified with having the option to revert the modification. This is done with a helper class `SDCNodeList` (for the sake of simplicity furthermore just called "node list"). The node list orders and allows reordering of the nodes, considering some requirements which are explained later. This is the base of the SDC scheduling appoach. The (mathematical) resource constraints for the linear program are created from this list where the order of the list is the elementary part of the SA/SDC scheduling approach.

The node list basically takes the the graph to schedule as constructor parameter. Then it initializes a corresponding `Node`-array with a length equals to the number of nodes in the graph. The original `Node`-class (from the high-level-framework) has got a new member `depth`, which is the hierarchical depth of each node (i.e. root-nodes have depth=0, their predecessors have depth=1 and so on). The `Node`-array will then be filled ordered by the corresponding depths. This leads to the initial node list which then can be modified in order to optimize the results.

To get better results, the list must be reordered somehow, but it must not do "wild swapping" of nodes - the reordering is restricted by the data flow dependencies. To clear this up a bit, figure 4.1 shows the example graph from the lecture.

The initial order of the node list (due to the hierarchical depth) could be #1, #2, #4, #7, #8, #3, #6, #10, #11, #5, #9. Node #3 e.g. is a data flow dependency of #1, #2, #5 and #9. Exactly these dependencies must be considered when #3 is moved in the list. To be more precise, each predecessor of #3 must always be left of #3 and each successor must always be to the right. This restricts the degree of flexibility when modifying the list.
To make modification easy, the `SDCNodeList` provides two methods `shoveRight` and `shoveLeft`, which do the following:

```
// nodes is a class-member
shoveCount: int
for(int i := i0+1; i < nodes.length; i++) {
  // find first non-flow-dependend-node
  if (nodes[i] isNoSuccessorOf nodes[i0]) {
    shoveCount = i - i0;
    break;
  }
}
// move items [i0 ... i0 + shoveCount] one array i0 to the right
// set original item from list[i0] = list[i0 + shoveCount]
```

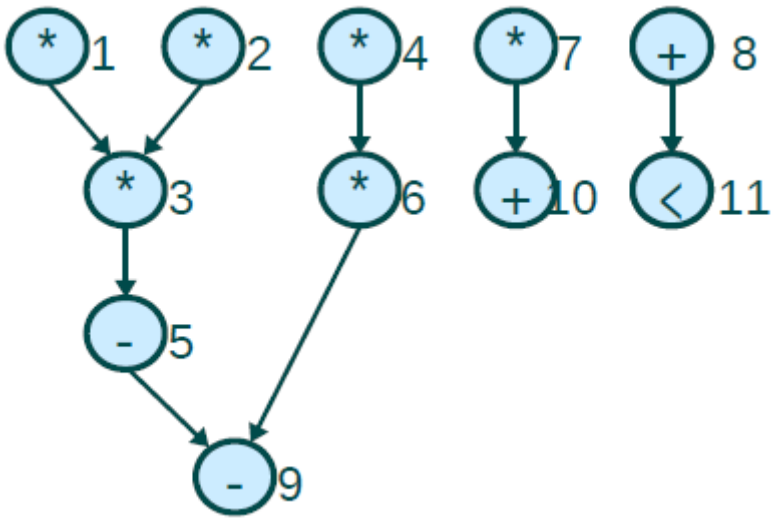

Figure 4.1: Example Graph ([2], Chapter 3, Slide 97)

This is a generic example and works in almost the same manner for shoveLeft. Basically the shove-functions find the first non-dependend predecessor/successor node beginning searching at `i0`. The initial ordering in the example before ended with the nodes #11, #5, #9. When shoving #9 to the left, the first node which would be checked to be predecessor is #5. Next iteration, #11 fulfils the *isNotPredecessor*-criterion, therefore the `shoveCount` would be set to 2 and the loop cancels. Now the elements would be reordered to #5, #9, #11. Long story short, when left-shoving #9, #5 must also be shoved, to stay before #9.

By storing the `i0` and `shoveCount`, the operation can easily be reverted if the change is not accepted the SA algorithm.

4.2 The SASDC-scheduler

After having a node list with valid ordering the scheduler can use this list to create a linear program which can be solved by the lpsolve library (as part of the SCPSOlver library). From the min-solution of the linear program the schedule can be created which then can be used from the SA algorithm (in order to compare the costs of the current and the modified schedule).

The outline of the whole SA/SDC scheduling then is as following:

- Create node list (which will immediatly provide the base for the resource constraints)
- For N Nodes, perform N random modifications on the node list in order to find a reasonable start temperature
- Use the N-times modified node list as initial schedule for the SA algorithm

The modification of the node list is very simple: A random number $r \in [0..1]$ and from it results the shove-operation: $s = round((2r-1)*(N-1)), \mid s \mid \in [1, N-1]$. sgn($s$) is the direction ($s < 0$ means shoving left, $s >= 0$ means shoving right) and $\mid s \mid$ is the index in the node list, which node should be shoved. The `shove`-methods return a flag indicating whether the shove was possible or not - if it was not possible, another random shove is tried, until it worked. The first and last nodes are not considered, because of the 50% likelihood that a shove would not be possible (e.g. the first element never be shoved left).

Another method `makeSchedule` does the whole work of creating the linear program from the node list, solve it and then create and return a schedule.

Now everything is prepared for running the SA algorithm. Another word on the linear program: It is not necessary to re-create a new `LinearProgram`-object for each schedule: The linear program can be created initially with the data flow constraints and can then be reused. When modifying the node list, only the "old" resource constraints in the linear program must be removed and be replaced by the new ones. This probably gives little performance gain.

5 Evaluation

File	Number of Nodes	cost fkt of ASAP	cost fkt of ALAP	cost fkt of SA/SDC	Quality factor of SA	Number of Iterations	Runtime / s
ADPCMn-decode-271-381	27	64.8	56.0	31.9	1	20494	29.39
				30.5	5	82216	113.38
				30.5	10	144181	185.55
ADPCMn-decode-425-472	12	23.8	23.8	34.2	1	85	0.11
				21.8	5	7591	6.37
				21.8	10	11001	9.13
ADPCMn-decode-524-553	7	25.4	26.4	25.4	1	15	0.02
				25.4	5	68	0.07
				25.4	10	135	0.13
ADPCMn-decode-559-599	9	26.1	27.2	28.2	1	20	0.03
				25.1	5	4137	3.26
				28.2	10	377	0.35
ADPCMn-decode-631-729	23	44.8	40.0	33.3	1	5017	5.81
				24.3	5	13121	14.65
				22.9	10	108076	120.56
ADPCMn-decode-771-791	5	13.5	13.5	22.5	1	10	0.01
				22.5	5	44	0.04
				22.5	10	87	0.08
ADPCMn-decode-803-832	8	18.8	18.8	20.2	1	17	0.02
				20.2	5	81	0.08
				20.2	10	161	0.16
AESrkgcyclic	24	24.4	23.4	37.4	1	421	0.61
				32.4	5	13881	18.84
				30.4	10	28414	38.14
BLAKE256Digest-processBlock-160-230	21	57.4	58.4	32.4	1	10905	12.11
				22.4	5	58291	64.11
				31.4	10	175161	194.06
BLAKE256Digest-processBlock-189-1577	308	414.9	128.7	110.9	1	6244	241.63
				97.9	5	31204	1236.87
ContrastFilter-filter-13-252	47	72.0	67.7	50.8	1	4251	9.49
				37.7	5	24622	54.00
				33.2	10	101821	218.61
ECOH256Digest-AES2RoundsAll-2-666	179	262.3	174.4	58.1	1	175567	2733.60

The table above compares the results of simple ASAP / ALAP-Schedules with the results of the implemented simulated annealing algorithm. Not all available datasets were computed due to long computation times. For some examples, the temperature converged extremely slow.

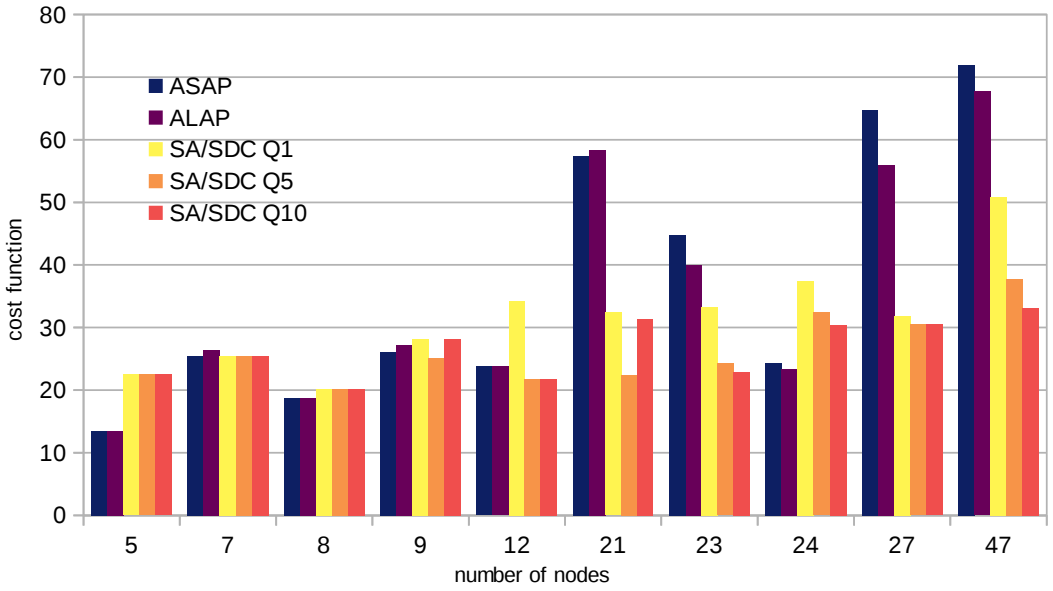

Figure 5.1: Comparison of cost

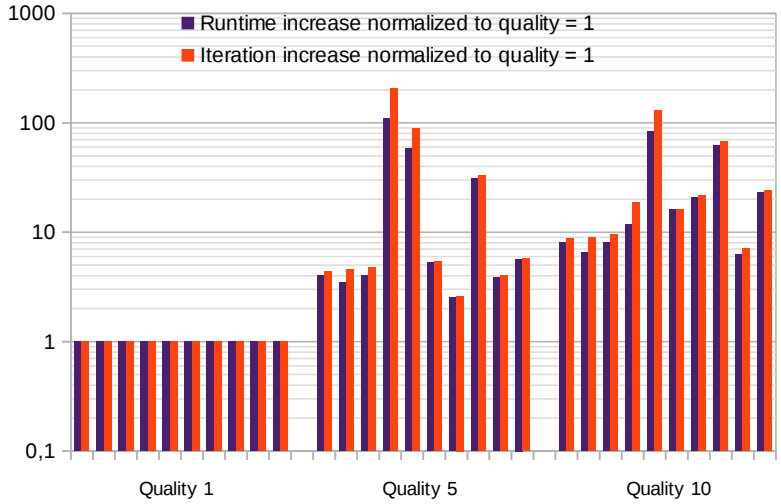

Figure 5.2: Comparison of runtime

It can be seen, that the SA-approach decreases the cost of large schedules, while the benefit for small schedules is comparatively low or in some cases even worse than the result of ASAP or ALAP.
The results of ADPCMn-decode-559-599 (9 nodes) or BLAKE256Digest-processBlock-160-230 (21 nodes) show an other anomaly: The result of quality 5 is better, than the result of quality 10, while the runtime of quality 10 is lower. This may be triggered by the initial random configuration, which is non-deterministic in the regarded implementation.

6 Conclusion

First things first: It is difficult to decide whether the results are good or at least reasonable. The main problem here is that we have no other results for comparison, neither for an "acceptable" runtime nor for the minimum cost for larger graphs in relation to the ALAP and ASAP results.

In order to optimize the scheduler (in terms of runtime), there one should consider three main bottlenecks:

- To respect the "sub-order" in the node list when shoving, the node list performs the predecessor/sucessor-checking over and over again. These recursive calls are currently implemented as recursive DFS, which might be expesive on huge graphs. On figure 4.1 a DFS upwards from #3 would search the whole subtree before #1, even if it might just check against #2. Flow dependend nodes are always ordered by their hierarchical depth, and therefore immediate successors/predecessors are also the first occurences which can be found when shoving. Therefore a BFS might be more suitable.
- The next bottleneck is the permanent re-checking of the predecessors/sucessors itself. It would be very helpful to find some kind of code, which makes this check much faster. The most intimating solution would be to store all successors of a node, not only the immediate ones, which obviously would raise the memory-usage significantly. Better would be some kind of numeric encoding.
- Another challenge is the optimizion of the SA itself: It is possible to often get the same solution which is sometimes accept it, sometimes not. Probably it is reasonable to somehow store the already generated nodelists (e.g. as hash) and restrict the usage of them: this would reduce the number of recaluclations (esp. lp_solve-calls)

Bibliography

[1] Jason Cong and Zhiru Zhang. *An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation*. University of California, Los Angeles, USA, 2006.

[2] Christian Hochberger. High-level synthesis (lecture). 2017.