Scheduling in a Real-time Network-on-Chip

Period minimization using metaheuristics

Mark Ruvald Pedersen Jaspur Højgaard Rasmus Bo Sørensen

Technical University of Denmark, Kgs. Lyngby

{s072095, s072069, s072080}@student.dtu.dk

Abstract

This paper shows how metaheuristics can be applied to optimize scheduling of of inter-processor communication in a real-time Network-on-Chip. This scheduling problem is an NP-complete multi-commodity flow problem, which we optimize using the metaheuristics Adaptive Large Neighborhood Search (ALNS) and Greedy Randomized Adaptive Search Procedure (GRASP).

General Terms Static scheduling, Metaheuristics, Network-on-Chip, Real-time, Multi-commodity flow problem

Rasmus

1. Introduction

This report is part of the work carried out in the course 42137 OPTIMIZATION USING METAHEURISTICS. Each section heading is annotated with margin notes indicating authorship. All three authors has contributed equally to this project and report.

The goal of this project is to implement a scheduler for static inter-processor communication in a real-time Multi-Processor System-on-Chip (MPSoC), the scheduler should use metaheuristics for optimization of a greedy solution. In this type of Real-time system scheduling is done at compile-time to simplify Worst-Case Execution Time (WCET) analysis. In real-time systems, performance depends purely on the Worst-Case Execution Time (WCET) – therefore the analyzability of the system is very important to obtain good performance.

An MPSoC is built of *tiles* consisting of a processor, a router and links to neighboring tiles. A sample sketch of a tile can be seen in Figure 1. In a real-time MPSoC, each tile processor executes one task to keep the timing analysis simple and get a lower WCET bound.

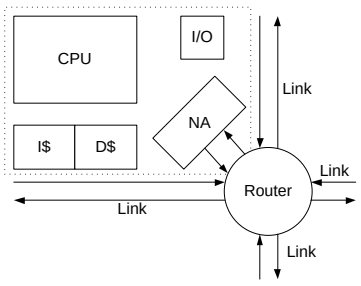
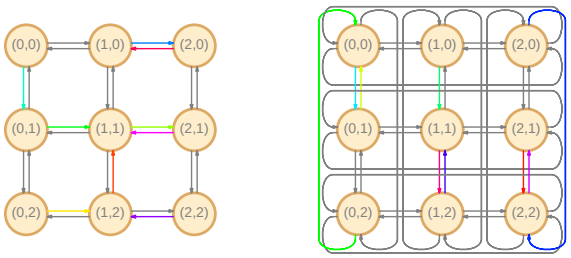


Figure 1: A tile contains a processor, a Network Adapter (NA), a local instruction and data cache and possibly some I/O.

An inter-processor communication channel is a point-to-point connection from one tile to another, this point-to-point connection can route packets along different paths. A path is the sequence of links, on which the packet travels to reach the end-point of the connection. Communication is statically scheduled such that communication on two channels can not interfere with each other – the communication channels are decoupled. Decoupling is required to make the NoC-interconnect analyzable with respect to WCET. The schedule is periodic, and thus for performance reasons (bandwidth and latency), we optimize for the shortest possible schedule period. Generating the optimal routing schedule is a hard problem to solve, for network topologies of a certain size generating an optimal schedule becomes impossible on the computer systems of today. The scheduling problem, has multiple sources and sinks making it a multi-commodity flow problem. The nature of the problem does not allow fractional flows on edges, thus the multi-commodity flow problem becomes NP-complete[ref: Karp]. In Figure 2 we show a example of the first time slot of a 3×3 mesh schedule generated by our scheduler, the figure is auto generated by the scheduler.



(a) Mesh (b) Bi-torus

Figure 2: The first time slot of the schedule. The color indicates which communication channel is routed on the link.

Our goal for this project is to make a scheduler that makes as good schedules as possible within a reasonable amount of time. The scheduler is meant to be usable for research purposes, thus we have constructed it very generic. Our scheduler can schedule arbitrary communication graphs on arbitrary interconnect topologies, the topologies are limited to routers with out degree 5, where one of the output ports always are connected to a CPU.

It is trivial to construct a greedy solution for the scheduling problem. To improve schedules compared to this greedy solution, we can apply metaheuristics.

Our scheduling problem is large and combinatorial in nature with both dependencies and constraints. A natural constraint is that any directed link can only transfer a single packet at any time-slot. Another constraint is that packets must arrive in the same order as they were sent. To avoid excessive feasibility-checking of the schedule due this in-order requirement of packets, we make the

simplifying decision that all paths should be a shortest path. Due to the regularity of the graph, this shortest path routing is simple. Always taking the shortest path will automatically guarantee causality of received packets.

Our scheduling problem is NP-complete, thus we need metaheuristics to improve schedule periods from greedy algorithms. For most relevant network sizes it is practically impossible to search though the whole solution space.

Rasmus

2. Metaheuristics

To construct at good solution we expect that a greedy algorithm will give fairly good results, to improve the greedy solution metaheuristics can be used. A given greedy solution might not be a good target for optimization, due to the high density of the solution. A less dense solution might lead to a better solution.

Our intuition on how a dense solution can be improved, tells us that it has to be spread out before it can be compressed further.

Mimicking nature we think of the free links in the network as air bubbles. We then need to make room for the air bubbles to surface, this can be done by decompressing the schedule. To conclude, we should not only accept better solutions than our current.

Jaspur

2.1 Initial solution

Four different approaches to create a feasible initial solution to optimize are:

Basic (BASIC) This initial solution creates a feasible schedule by randomly picking channels to be scheduled onto the NoC, and then schedules them on timeslots one after another. This solution only has one channel scheduled at each timeslot, and the advantage using this approach, is that the schedule is not dense and thus the applied metaheuristic will perform all the work towards an optimal solution.

Deterministic (GRREDY) The created deterministic approach is a greedy algorithm that tries to schedule channels onto the NoC at the earliest possible timeslot. The channels are scheduled in an order, where the longest channels are scheduled first. Length of channels is defined, by the number of links it has to take to reach its destination. The order, in which availability of links out of a router is checked, is also set. Due to the given attributes, this inital solution always gives the same result when given the same input. Therefore, using this initial solution, the metaheuristics are forced to try and optimize from the same starting point every time.

Randomized-routing (rGREEDY) This initial solution uses the same greedy approach as that of the deterministic, in that it tries to schedule the longest channels first, and the channels are scheduled at the earliest possible timeslot. The difference from the deterministic greedy algorithm is in how each channel is routed. Where the deterministic algorithm will always route the channels in the same way, the priority of possible available out links at each router is random, and thus if some routes can be routed better by using a different shortest path, this initial solution may find that route. Also, this gives an initial solution to the metaheuristic that is not always the same.

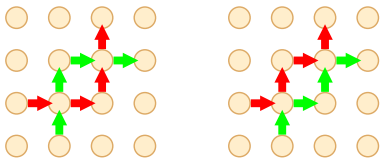
Randomized-routing and order (RANDOM) Here, again, the channels are scheduled as early as possible, but now the order is random in which the channels are scheduled. This is another initial solution, where the starting point for the metaheuristic is a somewhat dense schedule, but since the routing ordering is random, using this initial solution will explore solutions different than those when deterministic or only randomized routing is used.

The main motivation in having four different ways of obtaining initial solution is to explore different neighborhoods with having different starting points.

2.2 Neighborhood and operators

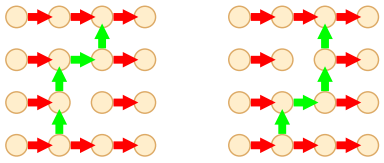
Mark

Our scheduling problem is to map a collection of channels to paths throughout time, without communication conflicts on any links at any time. A schedule with such conflicts is not feasible. The neighborhood is the search space around a current feasible solution when taking a small step away from it, i.e. incremental perturbations. Small perturbations of a current solution could be swapping pairs of links (Figure 3) or moving bends (Figure 4).



(a) From (b) To

Figure 3: Swapping pairs of links. This does not destroy feasibility, but has no advantage if both channels are present.



(a) From (b) To

Figure 4: Moving bends. This may not be feasible.

2.2.1 Primitives

Mark

Swapping pairs of links is always possible, but moving bends can lead to an infeasible schedule. Note that very little can be gained by swapping pairs of links – only if one of the channels of Figure 3 is absent may there be a slight advantage. As the bend-moving example hints at, there is very limited freedom of movement in dense schedules. Hence we expect most local perturbations lead to infeasible schedules. Even if such small local perturbations could make room for other channels to be scheduled earlier, these small changes alone are unable to change period of the schedule. So instead of taking small steps, we would rather take large jumps. Instead of swapping of pairs or moving-bends as our primitive functions, we consider removing entire already-scheduled channels (`rip-up`) and rebuilding (`route`) them. Given a channel, `rip-up` deletes the entire already-scheduled channel from the schedule. Given an unscheduled channel, `route` will greedily (making randomized shortest-path routing decisions) try to schedule the channel as early as possible. `route` does this via a simple depth-first search. Of the paths which has just been ripped-up, the longest paths are routed first.

This approach of rerouting entire channels is obviously a superset of the small local perturbations. Thus the rerouting-approach greatly increases our neighborhood and increases our chances of finding a new and feasible path. Thus our neighborhood of a current schedule is the space formed by how many and which channels we can reroute. The “intelligent” selection of which channels to reroute is the job of our operators.

Mark

2.2.2 Operators

Which element from our neighborhood to jump to next, is determined by our operators. Several operators have been implemented, as this is required to make LNS adaptive. Since our primitive functions are always `rip-up` and `route`, the operators are reduced to simply selecting which channels should be rerouted.

Dominating paths Recall that our main objective is minimize the schedule period. It is therefore natural to attack the paths which finish last, since they prolong the schedule. We call these paths for the *dominating paths*. However simply rerouting only the dominating paths, is likely to not improve the period: All paths are rerouted in a greedy fashion, starting as early as possible. We therefore consider more than just the dominating paths, we would also like to select the paths that prevents the dominating paths from starting earlier. An approximation of those paths preventing a dominating path P from starting earlier, is simply the paths below P over all time. I.e. we select all the paths that goes through a link before P goes through it. Even through this is a gross approximation, this operator has shown to be the most popular in ALNS.

Dominating rectangle Rather than just selecting the channels below P in time, we select all paths in the shortest-path rectangle containing all of P. This is written as a BFS search, only taking links which are on the shortest-path.

Late paths This is exactly like dominating paths, but also considering the paths finishing next-to-last, i.e. the paths finishing at time $p-1$ and $p-2$.

Random Simply selects a random set of paths. Also the number of selected paths is random: At least 2 paths are always selected, but up to 10% of all existing paths may be selected.

2.3 Proposed metaheuristics

We expect the following two metaheuristics to yield good results:

- Greedy Randomized Adaptive Search Procedure (GRASP): Make greedy randomized initial solution. Make local search for better solution. Start over with new initial solution.
- Adaptive Large Neighborhood Search (ALNS): Choose initial solution. Destroy and rebuild part of current solution. This is exactly what we described above.

Jaspur

2.4 Greedy Randomized Adaptive Search Procedure

With the nature of our problem, where there is not a clear neighborhood, diversifying the search for solutions may be an advantage. Also, the problem is very large, and these properties favor the GRASP metaheuristic.

The GRASP metaheuristic consists of the phases, greedy randomized adaptive phase followed by a local serch phase. The greedy randomized adaptive phase creates an initial greedy randomized solution adaptively. After creating the initial greedy randomized solution, the solution is improved iteratively using a local search. When the solution can not be improved locally, the best solution is stored, if it is the best seen so far. These phases of creating an initial solution and performing a local search are then repeated for a set period of time.

Our use of the GRASP metaheuristics is not completely in correspondence with its definition. The adaptive function that is to be used when constructing the initial greedy solution is not present in that phase, instead some adaptiveness in selecting operators when performing local search is added. Also, the iterative improvement in the local searchonly performs one step in our case.

Instead of building the initial solution randomly and adaptively, the initial solution is built using a cross between approaches randomized routing and randomized routing and order. This is done by constructing a sorted list of the channels from longest to the shortest as done in the randomized routing approach. A float is then passed as parameter to our GRASP, that is between 0 and 1. It is normalized according to the number of channels to be scheduled. This number, is the number of times two channels are to be chosen in the sorted list and switched thereby partially ruining the sorted order of the channels. We may loop upon this float input as a variation of β only, instead of being a number between 1 and number of candidates, it is fractional and then normalized. Setting β to zero will mean that the initial solution used is that of the randomized routing.

The local search utilizes the operators dominating-{path,rectangle} and late-paths. One of these operators is chosen randomly according to a weighted random choice. After running the operator, the returned set of paths are ripped up and rerouted according to the same greedy appoach as the randomized routing. The weights w of each operator vary adaptively on how they improve the initial solution. The operators start out equally weighted, and at each iteration that they are chosen they multiplied with the value

$$\frac{[Schedule\ length\ after\ initial]}{[Schedule\ length\ after\ local\ search]}$$

This means that dependent on how much an operator improves an initial solution, it is going to be that more favored to be selected again.

2.5 Adaptive Large Neighborhood Search

Mark

Contrary to GRASP, ALNS continually works on the same solution. ALNS repeatedly destroys a part of a solution and rebuilds it again into a feasible solution. Since we always rebuild with route, what we punish and reward are the selection operators described above. The operators fit into this scheme perfectly, so our implementation of ALNS is simple. Our ALNS implementation is as follows:

1. An initial solution is constructed using one of the specified algorithms described in Section 2.1. This is specified by the user.
2. One of the selection operators described in Section 2.2.2 is chosen stochastically. Each operator has an associated probability of it being chosen. To achieve some kind of Simulated Annealing-like behavior we give the random-selection operator a higher probability initially: Initial probabilities are 33% for the random operator and 22% each for the rest, dominating-{path,rectangle} and late-path. Since we have not performed any kind of extensive parameter tuning, we do not know how much these initial probabilities affect the solution.
3. The selection operator is executed, which returns a set of paths. This set of paths is ripped-up and re-routed again. As mentioned before, re-routing paths is done greedily: We first sort the ripped-up paths by length and route them according to this ordering. Any path is routed in a first-fit fashion, meaning we schedule it as early as possible. Also note that tie-breaking for shortest-path routing decisions is done stochastically.
4. The new schedule length is found, and if it the shortest seen until now, we copy it and remember it as our best solution thus far.
5. The selection operator we chose in step 2 is punished or rewarded according to a simple scheme: We calculate the ratio r between the period of the previous schedule and the new current schedule; $r = prev/curr$. Thus multiplying the probability of selected operator by r will automatically punished or reward it. If $r > 1$ the current solution is shorter than the previous, so we

reward by increasing the probability of the same operator being selected in the next iteration. Experimentation showed that dominating-paths was much better than random, so probability of random-selection drops quickly. To combat quick convergence we actually multiply by $\sqrt{r}$ rather than r.

Mark

3. Implementation

Our scheduler is written in C++11, using BOOST 1.49[1] and pugixml 1.0[2] as 3rd party libraries. The source code of our schedule is provided as open source software. The source is available at https://github.com/rbscloud/SNTs. The source code builds successfully on Linux and Cygwin with GCC 4.5.2 or later with `--std=c++0x` flag enabled.

The rest of this section will remain language agnostic.

3.1 Data-structure organization and delta-evaluation

In our problem domain, there are basically two ways to represent the schedule; time-centric or network-centric:

Time-centric We could consider grouping all routing-decisions made in the same timeslot, into a frame. The entire schedule would then consist of a layering of such frames, numbered from 0 through $period - 1$. We would have to be able to index into a frame given a router and its link, asking whether any channel in this frame has been scheduled on the link. However the interconnection network can have holes, and could be sparse.

Network-centric In reality, links are resources for which communication must contend. Thus is its more intuitive to have each link and router represented once only – rather than multiple times in each frame. This means we localize the schedule, storing a local schedule on each link. In this representation, this means we do not have to go up one timeslot and index again to check if something has been scheduled – this information is local. Splitting the schedule up into such localized schedules, also allows us to make fast successor queries. A drawback is that determining global schedule length now takes $\mathcal{O}(n)$ time where n is the number of routers, rather than $\mathcal{O}(1)$ time as in the time-centric representation. However using delta-evaluation this is reducible to average-case $\mathcal{O}(1)$ time.

The goal of delta-evaluation is to reuse as much of what is known about the previous solution, so the evaluation function becomes quick. In our case, we could store a dynamically re-sizable array which for each index (representing a timeslot), stores how many links is scheduled in that timeslot. Whenever we mutate our schedule, the array is updated correspondingly. Traversing backwards from the end of the array until hitting a non-zero, reveals the schedule length. Since the period of neighboring schedules is quite close to each other, traversing the array will take $\mathcal{O}(1)$ time in the average case, but $\mathcal{O}(p)$ time in the worst-case. Finally, resizing the array dynamically takes amortized constant time.

In reality, the overall performance depends on channel-routing and very little on our evaluation function, which is very simple and quick enough. Hence we have chosen the more Object Orientation-friendly network-centric representation. If further performance improvements are sought, our `route` primitive can be easily parallelized.

3.2 Calculating shortest paths

One of our simplifying decisions is to only route channels along shortest paths, as this will automatically guarantee in-order packet arrival. For this, we need to compute local routing decisions for the all-pair shortest paths. Since our network graph is very regular as each router only has max out-degree 4[3] and all links have unit cost, we can easily find the shortest path via a breadth-first search: We simply start at the destination b and do a BFS – as we encounter another router a, we now know a shortest path from a to b. That the path is a shortest path is guaranteed by the nature of BFS. Encountering a, we simply store which port the BFS-wave reached a from, and associate b with that port in a local (perfect) hashmap, $next$. So $a.next[b]$ returns a set of output ports of a, which lie on one of the shortest paths towards b. Computing all these shortest-paths can be done in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space.

[1] http://www.boost.org/doc/libs/1_49_0/

[2] http://pugixml.googlecode.com/files/pugixml-1.0.zip

4. Parameter tuning

Jaspur

Parameter tuning is vital, in order to get the most optimization from the metaheuristics. In our case the main parameters to consider were:

Running time Is the time that is given to the metaheuristic to try and optimize the problem

β The parameter determining the randomness of the initial solution when running GRASP

Starting parameter of choose table The priorities of operators chosen during the

Initial solution The choice of what initial solution is chosen when running ALNS

The tuning was done mainly based on observations from test runs. During testing, we chose to focused on the mesh and bi-torus topologies and the scenario where each router communicates a packet of length one to every other router in the network. We focused on the mesh and bi-torus with height and width 3-10 and 15. Also, there was not enough time for thorough testing, so possible scenarios such as running a metaheuristic on very large networks for e.g. 24 hours for getting test results to then do reruns was not an option.

4.1 Running time

The time, the metaheuristic needs to optimize a problem depends on the size of the problem at hand. Table 1 presents the number of iterations during a test run for two hours of each considered scenario.

Size	ALNS (RANDOM)		GRASP ($\beta = 0.1$)	
	Bi-torus	Mesh	Bi-torus	Mesh
3×3	466.451.877	271.864.914	33.164.408	30.629.712
4×4	125.812.628	100.497.005	6.997.571	5.486.080
5×5	87.230.064	29.898.319	2.555.017	1.886.940
6×6	39.682.545	10.825.917	823.677	457.601
7×7	19.926.148	4.270.345	247.464	159.557
8×8	6.382.482	1.478.727	123.695	55.597
9×9	3.205.896	1.478.727	64.213	21.638
10×10	1.350.495	353.699	27.495	11.543
15×15	15.969	6.944	1.070	84

Table 1: Rounded numbers of iterations when running for two hours on gramme.ebar.dtu.dk

When we look at the values in table 1, we see that the number of iterations for alns

We did not have time, to do test runs for a longer period of time.

[3] For routing we only care about the 4 non-local ports of routers.

4.2 Starting parameter of choose table

The starting parameters of the choose table denote the starting priorities of the available operators for the metaheuristic. The nature of the operators dominating-{path,rectangle} and late-path implies that they should be given equal weight at the start of optimization. The adaptive function will then favor those that give good results and not the others. We have not tested other scenarios than equal weighting of the starting parameter in the choose table. The random operator though, which is only present in ALNS, we assigned different values. The reason is that it is not very likely that the random operator is going to increase the

We ended up setting the starting parameter for the random operator to 1.5, while the others are set to 1.0.

4.3 β

When running optimization using GRASP, we tested the algorithm by running for two hours with

$$\beta \in \{0, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.5, 0.9, 1\}$$

Varying the β did not seem to matter that much if it was between 0 and 0.5. When running for two hour, almost all the values were the same. As the width and heigth increased to 10 and 15 though, a β less than 0.1 gave the best results. We also observed the best results for a bi-torus with size 8×8 with beta equal to 0.1. From this single case, where 8×8 performed better than the rest, we could consider 0.1 as about optimal for β. This is still close to the greedy solution, but leaves som room for randomness, that may improve the solution.

4.4 Initial solution

When considering ALNS and the initial solutions used, we do not consider average value of the schedule length. This is because as already stated, the performance of a real-time system is based on the worst case execution time period and thus we are most interested in the initial solution giving the best solution. Also, the length of the initial solution and the first couple of iterations will be worse for the BASIC initial solution than for the others

When doing a test run for 2 hours, the found schedule length for the different sizes of the bi-torus and mesh topologies are seen in table 3 and 3.

Bi-torus	BASIC	GREEDY	rGREEDY	RANDOM
3×3	11	11	11	**10**
4×4	**19**	**19**	**19**	**19**
5×5	**30**	**30**	**30**	**30**
6×6	**45**	**45**	**45**	**45**
7×7	**63**	**63**	**63**	64
8×8	87	87	**86**	88
9×9	118	**113**	115	119
10×10	160	**153**	155	158
15×15	514	**471**	477	507

Table 2: Best found period for bi-torus, when running ALNS on different initial solutions and sizes of networks for two hours

Generally for the biggest networks, the GREEDY algorithm seems to be the best to use as an initial solution. This is due to fact, that as the size of the network increases, the schedule becomes more dense, and thus the schedule produced initially by the deterministic greedy algorithm is the best found schedule.

Rasmus

5. Results

Our scheduler has been tested with custom topologies besides mesh and bi-torus, to verify functionality. For evaluating the performance

Mesh	BASIC	GREEDY	rGREEDY	RANDOM
3×3	**11**	**11**	**11**	**11**
4×4	**21**	**21**	22	**21**
5×5	**39**	**39**	**39**	**39**
6×6	**65**	**65**	**65**	**65**
7×7	102	98	102	**97**
8×8	152	**144**	**144**	145
9×9	217	**201**	205	202
10×10	299	**271**	275	**271**
15×15	1022	**886**	907	900

Table 3: Best found period for mesh topology running ALNS for two hours.

of our metaheuristics we only run our scheduler on use-cases with all-to-all communication and on either a mesh or a bi-torus topology these are the only examples for which we have results to compare with. For all-to-all communication in a mesh or bi-torus topology we also have theoretical lower bound on the scheduling period.

In [Schoeberl2012] lower bounds for the schedule period are given. In the article [Brandner2012] a heuristic scheduler is proposed, this scheduler has the constraint compared to our scheduler that the scheduler in each router must be the same. This is done to simplify hardware. We compare the results of our scheduler to the heuristic scheduler of [Brandner2012] in table and the lower bound

All our runs of our metaheuristic ran two hours.

To evaluate the how good our metaheuristics work in the

Link utilization: The schedule becomes more dense as the network size increases.

Iteration count graph

6. Conclusion

A. First appendix

Appendix, only when needed, can also be started behind the bibliography.

References

[1] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. I*n Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, July 2002.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

Size	Mesh					Bi-torus				
	Bound	[Brandner2012]	GREEDY	ALNS	GRASP	Bound	[Brandner2012]	GREEDY	ALNS	GRASP
3×3	8	28	13	**11**	**11**	8	11	12	**10**	**10**
4×4	16	59	24	22	**21**	15	20	21	**19**	**19**
5×5	25	112	41	39	**37**	24	**28**	32	30	30
6×6	54	–	66	65	**61**	35	–	45	45	**43**
7×7	66	–	98	97	**94**	48	–	64	63	**61**
8×8	128	481	144	143	**139**	64	88	87	86	**85**
9×9	135	–	201	200	**196**	90	–	**113**	**113**	**113**
10×10	250	974	271	271	**267**	125	158	154	153	**151**
15×15	600	3467	**886**	**886**	900	420	481	**471**	**471**	474

Table 4: Results compared to the heuristic results of [Brandner2012].