Weekly Report for Yu Hu's work in week8

March 6, 2005

In this week, I focused on the following four works:

1. Add min-delay pruning in buffer insertion package in BIC (We've discuss it in e-mail). Details are in section 1.
2. Make a brief analysis about the things behind the experimental results (We've discuss it in e-mail). Details are in section 4.
3. Expand King Ho's sampling to **3-dimension sampling** (sampling with power, RAT and capacitance). Experimental results show that 3-D sampling can achieve up to 5 times speedup upon 2-D sampling. Up to now, we can obtain up to 20 times speedup upon King Ho's code for single-Vdd buffer insertion. Details of 3-D sampling are shown in section 2.
4. Integrate buffer insertion technologies into tree construction package (route_determ) in BIC. Experimental results show we achieve up to 50 times speedup upon King Ho's code for single-Vdd buffer tree construction. Details are in section 3.

1 Min-delay pruning

I've added the predictive min-delay pruning rule ($\alpha(p, c, rat)$ is pruned, if $rat - minDelay < RAT*$) into BIC package and obtained about 2 times speedup over my non-min-delay version for single-Vdd buffer insertion.

In my calculation, I set the unitLenMinDelay = 0.02ps/um for 65nm node in BIC package, and the solutions produced by min-delay pruning version have the same quality with King Ho's. So this can verify the effectiveness of min-delay pruning.

After that, I tried to modify the predictive power pruning in my last version as following rule:

"$\alpha 1$ can be pruned, if $p1 + pre - p1 > p2$ and $RAT1 + pre - d1 < RAT2$ where $pre - p1$ is the predictive upstream min-power and $pre - d1$ is the min-delay between source and the current node."

After testing by s1-s6, I found the above power-pruning could prune little more redundant options, and the overhead of table lookup even takes much more cpu time. So the total running time is even longer than the version without power pruning. In fact, the reason is, the rule $RAT1 + pre - d1 < RAT2$ always can't be satisfied, since $pre - d1$ is often much larger than $RAT1$ and $RAT2$.

In my calculation, I doubt the correctness of the setting for "unitLenMinDelay". I calculate this value as 0.09ps/um in matlab using King Ho's ISLPED paper formula, but I found 0.09ps/um always gives a much larger prediction for upstream delay, and led to a wrong result. I set unitLenMinDelay from 0.09ps/um to 0.02ps/m and ran my code. Until I adjust this value to 0.02ps/um, I can get the correct results. So I set unitLenMinDelay 0.02ps/um in my implementation. Anyway, there maybe something wrong in my settings in matlab code.

2 3-dimension sampling

The main idea of 3-D sampling is sampling with power, RAT and capacitance. Given a power-RAT-capacitance 3-D space, we divide each side of the bounding cube of all options into equal segments such that the entire power-RAT-capacitance domain are superposed by a grid. For each grid cube in the bounding cube, we retain only one option if there is any.

In experiments, I set the sampling number as 20 for both 3-D sampling and King Ho's 2-D sampling. Which means 3-D sampling will pick up much lesser samples than King Ho's 2-D sampling. The following table shows the comparison among three strategies with single-Vdd buffer insertion (four kinds of buffers under 65nm are in buffer library and the minimal unit length is set to be 0.02um/ps). We can find that we obtain up to 20 times speedup upon King Ho's code within 1% worse of solution quality.

In the following table, columns pre-"3D" use (3-D sampling (20) + aggressive pre-buffer pruning + min-delay pruning), columns pre-"2D" use (2-D sampling (20) + aggressive pre-buffer pruning + min-delay pruning), and columns pre-"KH" are the original code used in King Ho's DAC'05 paper. The first column of each kind of strategies is CPU time.

Table 1: Buffer insertion: 3-D sampling vs. 2-D sampling vs. King Ho

net	3D(s)	3D RAT(ps)	3D Pwr(fJ)	2D(s)	2D RAT(ps)	2D Pwr(fJ)	KH(s)	KH RAT(ps)	KH Pwr(fJ)
s1	0	-656.543	4689.09	0	-656.543	4689.09	3	-656.543	4689.09
s2	0	760.504	5631.62	1	760.504	5631.62	5	760.504	5631.62
s3	1	-1063.17	7891.46	2	-1063.17	7891.46	13	-1063.17	7891.46
s4	2	-948.835	12644.3	8	-946.501	12080.7	63	-946.501	12191.6
s5	6	-1811.46	19012.3	20	-1809.4	19011.3	166	-1809.15	19463
s6	10	-1722.43	25573.1	41	-1722.43	25461.2	270	-1722.43	25460.2
s7	24	-3050.83	35183.1	87	-3032.41	36308.4	575	-3028.05	36985.9
	1	$1 \pm 1\%$	$1 \pm 1\%$	4×	$1 \pm 1\%$	$1 \pm 1\%$	20×	1	1

3 Speedup in buffer tree construction

I've implemented my speedup technologies in buffer insertion, such as pre-buffer pruning (Weiping's), 2D sampling (King Ho's), min-delay pruning into buffer tree construction. I want to integrate my 3D sampling into tree construction, but the experimental results show much error after 3D sampling, so I'll consider why 3D sampling can't work for tree construction in the next week.

Besides, I integrated my grid reduction heuristic in tree construction package. About 2 times speedup is obtained by this strategy. The basic idea of grid reduction comes from the min-delay pruning.

3.1 My grids reduction idea

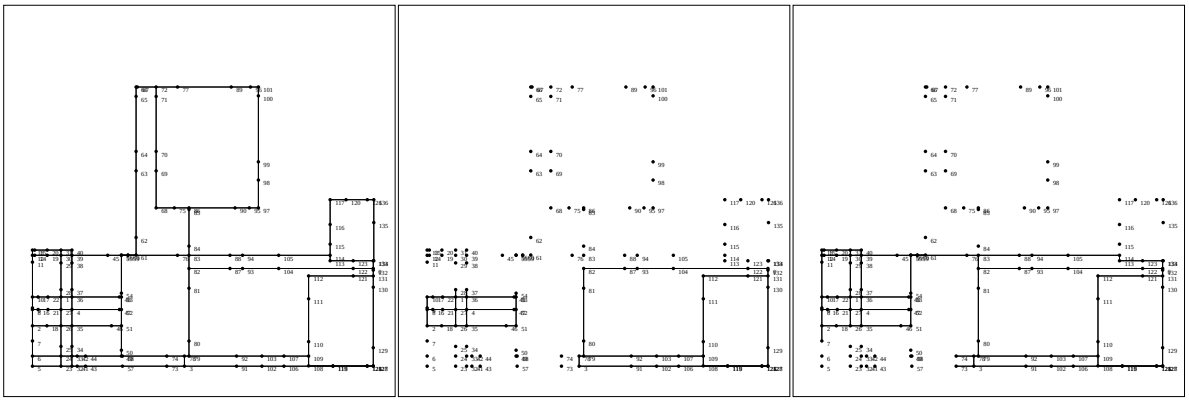
Given a grid node p, the min-distance from p to some sink s_1 is approximated as the Manhattan distance between them, denoted as (p, s_1). Similarly, the min-distance from p to source is approximated as $(p, source)$. To meet the delay budget, $(p, s_1) + (p, source) * unitLenMinDelay$ shouldn't exceed $RAT_{s_1} - RAT_{source}$. So we can savely delete those grid node p, which satisfying $(p, s_1) + (p, source) * unit - len - min - delay > RAT_{s_1} - RAT_{source}$.

However, I implemented this idea in my code, but failed to get a correct result. Since I can't get a correct $unitLenMinDelay$ value for test cases (grid.2 - grid.6), some non-redundant grid nodes are deleted, which leads to the failure to construct a completed tree.

Based on above idea, I just remain those grid nodes, such as p, which satisfying $(p, s_i)+(p, source) = (s_i, source)$, for any sink s_i. This rule means, we delete all grid nodes which are not in any rectangles formed by any sink-source pairs. This is reasonable in buffer tree construction, since a long-distance wire snaking won't be good for delay.

The main problem of this idea is that we might not get a connection graph after grid reduction. We can see this from Fig.1. Fig.1(a) shows the original grids of grid.4, and Fig.1(b) shows the grids after reduction. Obviously, Fig.1(b) isn't a connection graph, which means we can't obtain a tree connecting all sinks by this grids.

To tackle this problem, I relax the size of rectangle to graph_size/10 larger, which means the rule is modified as $(p, s_i) + (p, source) < (s_i, source) + graph\_size/10$. Using this rule, we can make sure of the connection for all test cases. The reduced grids for grid.4 is shown in Fig.1(c).



(a) (b) (c)

Table 2: Different grids for grid.4

3.2 Experimental results for buffer tree construction

The experimental results are shown in Tab.3, where columns pre-"GR" use (grid reduction + 2-D sampling (20) + aggressive pre-buffer pruning + min-delay pruning), columns pre-"O" use (2-D sampling (20) + aggressive pre-buffer pruning + min-delay pruning), columns pre-"KH" use King Ho's DAC'05 code. The first column of each kind of strategies is CPU time. We can see that we obtained 50× speedup upon King Ho's code by using "GR" strategy.

Table 3: Buffer tree construction: grid-reduce vs. non-grid-reduce vs. King Ho

net	GR(s)	GR-RAT(ps)	GR-Pwr(fJ)	O(s)	O-RAT(ps)	O-Pwr(fJ)	KH(s)	KH-RAT(ps)	KH-Pwr(fJ)
grid.2	0	-218.434	1107.07	0	-218.434	1107.07	0	-218.434	1107.07
grid.3	2	-557.398	3116.3	2	-557.398	3116.3	22	-557.398	3116.3
grid.4	1	-531.141	2636.54	4	-531.141	2636.54	42	-531.141	2636.54
grid.5	16	-523.429	3359.74	29	-523.429	3359.74	535	-522.448	3359.74
grid.6	24	-356.726	2819.41	38	-356.726	2819.41	1576	-356.726	2819.41
	1			2×			50×		

4 Discussions about experimental results for buffer insertion

In this section, I try to give a brief analysis about things behind my experimental results as follows.

Basically, several speedup technologies proposed in Weiping's DAC'03 paper are focused on timing optimization. These technologies are all performed on the searching tree.

As we know, in timing optimization, each option is a (rat, cap) pair, so all non-redundant options (their rat and cap can be sorted in the same order) in a node can be organized in one searching tree data structure. This makes the number of options in this tree data structure very large (usually over 100). So Weiping's sophisticated data structure added to the searching tree can show much effectiveness.

However, in power optimization, each option is a (power, rat, cap) triple, and obviously we can't sort power, rat, and cap with the same order as we do in timing optimization. So options can't be stored in a single searching tree in each node, instead, we store options with several subsets (each subset can be a tree or a vector) indexed by power or cap. This change makes the options in each subset much lesser than those in searching tree in timing optimization. This observation leads to ineffectiveness of Weiping's DAC'03 speedup technologies (the overhead of maintaining a complex searching tree overwhelms the tree's efficiency for small options number in the tree).

In fact, due to this situation, Weiping used only predictive pruning in cost-minimization buffer insertion in his ASPDAC'04 paper, in which the experimental results show that his method obtained less than 20 times speedup upon John Lillis' approach (for over 1k sinks net). Note that, King Ho's sampling can achieve 85X speedup for 800 sinks net, so Weiping's approach isn't so substantial.

Essentially, the most important thing to get speedup in power optimization is to reduce the options number as many as we can. So sampling can really do a good job, besides that, aggressive predictive pruning and min-delay pruning can also take much effect. These three methods make our code run much faster than Weiping's.

Based on the experimental results, we also get the following observation and conclusion.

In the cases smaller than 50 sinks, my method speedups about 2 times upon weiping's method. In larger scale cases, such as s5, s6, ..., my method speedups over 7 times upon weiping's. The main speedup comes from sampling (I set the sampling parameter as 20.) and our data organization method.

When the scale is small, many subsets contain less than 20 options, so sampling almost has no effect, while when the scale become larger, sampling can do a effective job for pruning.

Furthermore, our organization of options isn't quite the same with weiping's. The main different is, we index options with capacitance, and weiping indexes options with power. In my experiments, I found that there're much more different power values than different capacitance values, which makes the weiping's method spend more running time in exploring more subsets indexed by power.