On Mismatches Between Incremental Optimizers and Instance Perturbations in Physical Design Tools*

Andrew B. Kahng and Stefanus Mantik

UCLA Computer Science Dept., Los Angeles, CA 90095-1596
{abk,stefanus}@cs.ucla.edu

Abstract

The incremental, "construct by correction" design methodology has become widespread in constraint-dominated DSM design. We study the problem of ECO for physical design domains in the general context of *incremental optimization.* We observe that an incremental design methodology is typically built from a *full optimizer* that generates a solution for an initial instance, and an *incremental optimizer* that generates a sequence of solutions corresponding to a sequence of perturbed instances. Our hypothesis is that in practice, there can be a *mismatch* between the strength of the incremental optimizer and the magnitude of the perturbation between successive instances. When such a mismatch occurs, the solution quality will degrade – perhaps to the point where the incremental optimizer should be replaced by the full optimizer. We document this phenomenon for three distinct domains – partitioning, placement and routing – using leading industry and academic tools. Our experiments show that current CAD tools may not be correctly designed for ECO-dominated design processes. Thus, compatibility between optimizer and instance perturbation merits attention both as a research question and as a matter of industry design practice.

1 Introduction

With the shift to deep-submicron processes and high-performance constraint-dominated designs, achieving satisfactory design solutions has become increasingly difficult. Thus, "construct by correction" iterative design methodologies have come to replace "correct by construction" flows. One indicator of this shift is in commercial EDA tools, where placement-based synthesis [10], "in-place optimization", "routing-driven synthesis", etc. all support performance-driven incremental netlist and layout optimization. A typical iteration consists of (1) placement, (2) timing analysis, and (3) netlist reclustering, repeater insertion, and driver resizing; the netlist changes are then fed back to the placement tool as *ECOs* (*E*ngineering *C*hange *O*rders), and the sequence of steps (1)-(3) repeats. A second indicator of this shift lies in the architecture of future tool offerings, e.g., integrated RTL floorplanning and place-and-route tools that share a common design data repository for physical context (physical design library, chip layout plan), logic design (gate-level netlist), and analysis results (obtained by an "analysis backplane" of parasitic estimation, delay calculation, and incremental static timing analysis). Such a tool architecture allows backward loops between arbitrary stages within the design flow. For example, from detailed routing one might be able to iterate back up to placement, back up to logic synthesis, or even back up to RTL design, depending on the level at which changes are deemed necessary to simultaneously achieve routing completion, timing and signal integrity.

*This work was supported in part by the MARCO Gigascale Research Center and by Cadence Design Systems, Inc. Andrew B. Kahng is now Professor of Computer Science and Engineering, and of Electrical and Computer Engineering, at the University of California, San Diego.

We observe that *incremental optimization* lies at the heart of the iterative, "construct by correction" paradigm. EDA tools that support this paradigm will typically provide the user with an "incremental mode", where the solution to a new problem instance is obtained by perturbing the solution to the previous problem instance. For example, given post-placement changes to the gate-level netlist (due to driver resizing and repeater insertion) a previous placement result might be used as the starting point for iterative solution (using low-temperature simulated annealing) of the placement problem for the new netlist. Similarly, given post-routing changes to net criticalities and topology directives (due to timing and noise analysis results) a previous routing might be used as the starting point for iterative solution (using rip-up and reroute) of the routing problem. A number of incremental physical design formulations are reviewed in [5].

2 Incremental Optimization

A simple and general model of incremental optimization is:

- An *original instance*, I_0, is solved by a *full algorithm* to yield solution s_0.
- Perturbed instances $I_1,\ldots,I_n$ are generated one by one in sequence.
- Each perturbed instance is solved by an *incremental algorithm* which uses s_{i-1} as the starting point for finding solution s_i, $i = 1,\ldots,n$.

In what follows, we assume that both the full and incremental algorithms perform *iterative optimization* (see the definition in [3]), i.e., they iteratively generate a new solution from the current solution. If the algorithm can start the optimization process from any given initial solution, then incremental optimization is always possible with this algorithm. We also assume that each algorithm returns a *local minimum* solution.[1] Our work is motivated by the observation that the *full algorithm* is often quite different from the *incremental algorithm.* For example, a full algorithm for row-based placement might use a meta-heuristic combination of analytic placement and annealing, while an incremental algorithm might use only annealing.

2.1 A Potential Mismatch: Instance Perturbation vs. Strength of Optimizer

We ask the following question:

[1]This is a general model that encompasses greed, simulated annealing, and other heuristics. (1) A given instance is defined by a finite solution space S and a cost function $f : S \rightarrow \Re^+$. (2) The generation of new candidate solutions from the current solution corresponds to a *neighborhood structure* (i.e., a topology of "possible next-moves") over S, with $N(s)$ denoting the set of neighbors of solution s. (3) The cost function and the neighborhood structure together define the *cost surface* for the instance. (4) We seek a *global minimum* $s^* \in S$ such that $f(s^*) \leq f(s)\ \forall s \in S$. (5) A *local minimum* $s' \in S$ satisfies $f(s') \leq f(s)\ \forall s \in N(s')$.

Can the quality (say, relative to optimal) *of solution s_n be worse than that of solution s_0?*
And if so, can the decrease in solution quality be attributed to aspects of the sequence $I_0, I_1, \ldots, I_n$ (e.g., the "distance" between successive instances) and the incremental algorithm?

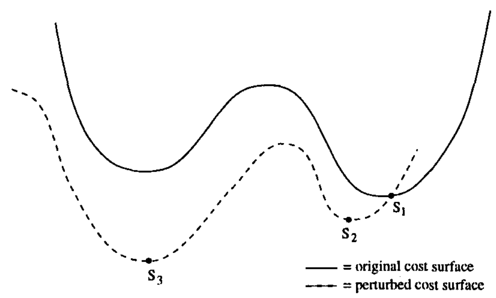

Figure 1: Perturbing the instance will perturb the cost surface.

(Of course, this question still makes sense even if the full algorithm is the *same* as the incremental algorithm.) The intuition behind our question is as follows (see Figure 1). Each perturbed instance I_i changes the cost surface of the optimization, e.g., from the solid cost surface to the dashed surface in Figure 1. The previous local minimum s_{i-1} (s_1 in the Figure) is in the "basin of attraction" [8] of some local minimum s_i (s_2 in the Figure) for the instance I_i, which is why the incremental algorithm returns s_i for this instance. But if the new instance I_i differs sufficiently from I_{i-1} – in particular, such that s_i is not a good solution (note the existence of solution s_3 in the Figure) – then the incremental algorithm must escape the "basin of attraction" in order to find a good solution. Our hypothesis is that the changes between successive instances must be *compatible* with the strength of the incremental algorithm:

- A *strong* incremental algorithm, in combination with *small* changes to the cost surface in successive instances, will maintain good solution quality but waste computational resources.
- A *weak* incremental algorithm, in combination with *large* changes to the cost surface in successive instances, will lead to progressively worse solution quality.
- In general, sufficiently large changes in the cost surface require sufficiently powerful incremental algorithms that escape stale basins of attraction in finding good solutions.

To the best of our knowledge, this question has not been addressed in the literature. The most closely related body of research is that on *problem-space* and *heuristic-space* methods in the metaheuristics literature [18] [12]. Such methods perturb a given instance to allow a given optimization heuristic to escape local minima. The perturbations induce alternate cost surfaces that one hopes are correlated to the original cost surface (so that good solutions in the new surface correspond to good solutions in the original), yet which have sufficiently different structure (so that the optimization heuristic can move away from the previous local minimum). While the success of such methods has been well-documented, no research has addressed the potential cost – in terms of either runtime or solution quality – of *mismatches* between perturbation size and strength of the optimization heuristic.

2.2 An Experimental Investigation

Our contributions stem from experimental analyses which clearly demonstrate that the mismatches noted above can be real.

- We define an experimental framework that allows us to find the maximum perturbation size that is consistent with no loss of solution quality as the incremental optimizer tracks the sequence of perturbed instances. This framework is used in three distinct domains: partitioning, placement and routing.
- We study incremental partitioning using a good partitioning tool. Our results show that applying large perturbations to the instance may allow the incremental optimizer to find much-improved solutions.
- We study incremental placement using a leading industry placement tool and a realistic model of netlist changes made for performance optimization. Our results show that current incremental optimization approaches, while quite strong, must nevertheless be carefully matched against the magnitude of netlist perturbations.
- We study incremental routing using a leading industry routing tool and a realistic model of routing changes for performance optimization. Our results show that the router is not well-suited for incremental optimization: a single drastic change with large perturbation size gives better results than an equivalent set of small changes with small perturbation size.
- We conclude that the *incremental optimization* model is of increasing importance in VLSI CAD applications, and that the question of potential mismatch between instance perturbations and incremental optimization engines merits further study.

In the following sections, we will describe our experimental methodology and results for hypergraph partitioning, standard-cell placement, and standard-cell routing domains.

3 Experimental Design

In this section, we describe an experimental framework for assessing the potential mismatch between (i) the incremental optimizer capability, and (ii) the magnitude of the instance change. We would like to understand (i) if applying the incremental optimizer to the perturbed instance will actually improve the overall solution, and (ii) if the size of the perturbation has any effect on the solution quality. To answer these questions, we have devised two simple experiments that test whether an incremental optimizer can maintain solution quality.

- **Reversal-Based Experiment.** We construct a series of instances $I_0, \ldots, I_{k-1}, I_k, I_{k+1}, \ldots, I_{2k}$ where $I_0 = I_{2k}$, $I_1 = I_{2k-1}$, etc. In other words, there are k successively perturbed instances $I_1, \ldots, I_k$, then we reverse the sequence so that the final instance I_{2k} is the same as the original instance I_0. With this experiment we can see whether the final solution S_{2k} has better quality than the initial solution S_0. Thus, this experiment shows us whether the incremental optimizer eventually improves or loses solution quality.
- **Dicing-Based Experiment.** We chop up a given perturbation into smaller perturbations. Then, we see whether applying the incremental optimizer to each of the corresponding perturbed instances in sequence results in better solution quality (or, better solution quality / runtime tradeoff point) than applying the incremental optimizer to the final perturbed instance alone. This experiment shows whether the magnitude of perturbation plays an important role in incremental optimization. If the size of the perturbation, ΔI, is too small and the incremental algorithm is too strong (a situation which always leads to a better solution), or if ΔI is too big and the incremental algorithm is weak (a situation which will lead to worse solutions), then there is a mismatch between perturbation size and algorithm strength.

We have performed these experiments in three distinct domains of physical design: partitioning, placement and routing. The detailed experimental setup for each domain is described in the respective sections below. Uniformly, we will use p to denote the magnitude of the perturbation, and k to denote the number of perturbed instances generated in the reversal-based experiment.

4 Partitioning

Our instances for partitioning domain are netlist hypergraphs. We use MLPart (a publicly available (academic) multi-level partitioner [4])[2] for both full optimizer and incremental optimizer.

Our instance perturbation consists of weight changes for hyperedges. Five discrete values (1, 2, 5, 8 and 10) are allowed for the hyperedge weights. Our initial instance is the original hypergraph with all hyperedges having identical weight = 5. In generating an instance I_{j+1} from instance I_j, we randomly select p hyperedges whose weights are to be changed. For each of these p hyperedges, we toss a coin to determine whether the hyperedge weight will be increased or decreased to the next value from its current value. If a hyperedge has the maximum (minimum) possible weight, incrementing (decrementing) will be ignored.

We start the reversal-based experiment by running the partitioner on original instance I_0 (again, with all hyperedge weights = 5) to obtain the initial solution S_0. The instance I_0 is then perturbed to obtain instance I_1. We apply the partitioner to I_1 with S_0 as the starting point; this yields solution S_1. We continue this process until we obtain the final perturbed instance I_k. From this point, we start to reverse the instance sequence, i.e., solution S_{k+1} for instance $I_{k+1} = I_{k-1}$ is obtained with S_k as the initial solution.

In the dicing-based experiment, we set up two independent runs of the partitioner. The first run creates a new instance I'_k from the original instance I_0 by changing the weights for p hyperedges all at once, then runs the partitioner to obtain S'_k. The second run creates a sequence of instances $I_1, I_2, \ldots, I_k$ that gradually implement the p weight changes, and with the final instance I_k being exactly the same as I'_k. The partitioner is called for each instance until we obtain the final S_k.

Experimental Results

We use four industry standard benchmarks from the IBM-internal circuits released in the ISPD-98 Benchmark Suite [1, 2]. Characteristics of these benchmarks are shown in Table 1. We run three sets of experiments with different p values: 14, 140 and 1400. Each run uses 10% for the partitioning balance tolerance, and actual cell areas for node weights. For the reversal-based experiment, we generate a sequence of ten perturbed instances ($k = 10$), i.e., we have 21 partitioning solutions $(S_0, S_1, \ldots, S_{20})$. Table 2 shows a representative sequence of perturbations with the corresponding results for the ibm04 test case. Table 3 shows results for all partitioning runs. Each value in this table is an average over 30 runs (maximum and minimum values, with runtime in seconds (Sun Ultra-1) in parentheses).

Design	# Terminal Nodes	# Core Nodes	# Nets
ibm01	246	12506	14111
ibm02	259	19342	19584
ibm03	283	22853	27401
ibm04	287	27220	31970

Table 1: Design test cases for partitioning experiments.

Instance	Solution	# Cut Nets		
		$p=14$	$p=140$	$p=1400$
I_0	S_0	497 (36)	518 (38)	507 (62)
I_1	S_1	489 (32)	508 (33)	498 (62)
I_2	S_2	487 (34)	504 (32)	495 (62)
I_3	S_3	482 (34)	499 (33)	494 (61)
I_4	S_4	481 (33)	497 (33)	493 (62)
I_5	S_5	480 (33)	496 (33)	493 (63)
I_6	S_6	480 (33)	492 (33)	494 (63)
I_7	S_7	480 (33)	492 (34)	493 (63)
I_8	S_8	479 (34)	493 (34)	493 (63)
I_9	S_9	478 (34)	492 (35)	495 (63)
I_{10}	S_{10}	476 (33)	483 (34)	494 (62)
I_9	S_{11}	475 (32)	482 (34)	495 (62)
I_8	S_{12}	474 (32)	482 (34)	493 (62)
I_7	S_{13}	474 (33)	483 (34)	491 (63)
I_6	S_{14}	472 (30)	482 (34)	489 (62)
I_5	S_{15}	473 (32)	481 (32)	488 (62)
I_4	S_{16}	472 (31)	481 (32)	487 (61)
I_3	S_{17}	472 (30)	480 (32)	487 (62)
I_2	S_{18}	472 (30)	480 (34)	486 (62)
I_1	S_{19}	471 (29)	480 (34)	485 (62)
I_0	S_{20}	471 (28)	480 (34)	484 (60)

Table 2: A sequence of perturbations and incremental optimizations for the ibm04 test case. Values are average net cut over 30 runs, with standard deviation in parentheses.

Design	# Cut Nets (CPU)			
	Initial	$p=14$	$p=140$	$p=1400$
ibm01	240	224/247/215 (9.21)	227/249/215 (9.25)	217/243/215 (9.21)
ibm02	303	283/304/250 (17.8)	286/314/249 (17.7)	286/365/249 (17.2)
ibm03	1016	676/757/631 (20.6)	665/807/629 (20.9)	672/791/635 (20.6)
ibm04	550	471/523/443 (23.5)	480/548/443 (22.7)	484/441/718 (22.9)

Table 3: Partitioning results for all test cases. Values are average/maximum/minimum over thirty runs, with CPU seconds in parentheses. Experiments are performed on a 170MHz Sun Ultra-1 machine and incremental CPU time is the total runtime for incremental mode.

In the dicing-based experiment, our first experimental run uses $p = 1400$, which changes 1400 hyperedge weights. The second experimental run uses the same hyperedge weight changes but with the overall perturbation broken down into 100 small steps, each with 14 weight changes. Table 4 compares results from both runs; again, values are averages over 30 runs.

Design	Regular Instance		Diced Instance	
	# Cut Nets	CPU	# Cut Nets	CPU
ibm01	253	10.85 secs	216	1082.44 secs
ibm02	268	20.03 secs	257	2012.73 secs
ibm03	718	24.55 secs	697	2366.12 secs
ibm04	520	26.37 secs	496	2597.42 secs

Table 4: Comparison between an incremental run on a partitioning instance with a single large perturbation, and incremental runs on the same instance with a "diced" perturbation.

Table 3 shows the improvement obtained by the instance perturbations. Incremental partitioning runs give a substantial improvement in overall solution quality[3]. This shows that **the problem-space metaheuristic approach may be more effective for partitioning.** We note that our conclusion holds even for runtime-equalized comparisons versus multistart execution of the multilevel partitioner. In general, re-

[2]See `http://vlsicad.cs.ucla.edu/GSRC/bookshelf/` and `http://www.openEDA.org` for free source codes. The partitioner can take an initial solution and perform the coarsening phase of multilevel Fiduccia-Mattheyses partitioning while respecting this solution (i.e., clusters must always group nodes from the same partition). However, in the uncoarsening phase the partitioning can diverge from the initial solution.

[3]Qualitatively similar results are obtained for flat FM (FMPart, also publicly available at http://vlsicad.cs.ucla.edu/GSRC/bookshelf).

sults of (S_{20}) runs are extremely good when compared with the best published solutions for these test cases.

5 Placement

Our instances in the placement domain are circuit netlists. The full optimizer is the Cadence QPlace 5.0.46 placer (all settings for maximum quality), and the incremental optimizer is the same QPlace placer, invoked in incremental mode. The initial instance is the original unplaced netlist, and instance perturbations comprise netlist changes which remove or insert cells and nets. To create instance I_{j+1} from instance I_j, we randomly select p cells and delete them from the netlist. If a net on which cells have been removed is disconnected as the result of the deletion, then it is merged with another net that is also disconnected from the same deletion. All dangling nets (i.e., nets that are connected to one cell only) are deleted. This perturbation (and its reversal) reflects ECO type operations used in performance optimization.

Again we start the reversal-based experiment by running the full QPlace placement on original instance I_0 to obtain the initial solution S_0. Instance I_0 is then perturbed by deleting p cells to obtain instance I_1. We apply incremental QPlace on I_1 with S_0 as the starting solution; this yields S_1. The process is continued until the last perturbed instance I_k is created, and from this point we reverse the instance sequence so that solution S_{k+1} for instance $I_{k+1} = I_{k-1}$ is obtained with S_k as the initial solution.

In the dicing-based experiment, we again set up two independent runs of the placer. The first run creates a new instance I'_k from original instance I_0 by making a big change in the netlist – i.e., p cells deleted at once – and incremental QPlace is called to obtain S'_k. The second run creates a sequence of instances $I_1, I_2, \dots, I_k$ that gradually implement the same perturbation, and with the final instance I_k exactly the same as I'_k. Incremental QPlace is called for each instance until we obtain the final S_k.

Experimental Results

We use two industry standard-cell benchmarks for experiments in placement domain; their characteristics are shown in Table 5.

Design	# Cells	# Nets
Test case 1	12133	11828
Test case 2	20577	25634

Table 5: Design test cases for placement and routing experiments.

We also use three different perturbation sizes (p): 12, 120 and 1000. Table 6 shows the results of all runs with different perturbation sizes. We again use $k = 10$ in this domain, which leads to 21 placement solutions $S_0, \dots, S_{20}$. For the dicing-based experiment, we delete $p = 1200$ cells: (1) the first run deletes all 1200 cells at once and then calls incremental QPlace to place the remaining cells, and (2) the second run breaks the perturbation into 100 steps with 12 cells deleted at each step, and incremental QPlace called for each step. Table 7 compares results from both runs.

Design		Initial	$p = 12$	$p = 120$	$p = 1000$
Test case 1	Wirelength	2796552	2726359	2724120	2736762
	CPU (secs)	190	1738	1655	1100
Test case 2	Wirelength	5928162	5748179	5757415	5758138
	CPU (secs)	409	4013	3852	3121

Table 6: Placement results for all test cases. Experiments are run on a 300MHz Sun Ultra-10. We see that incremental placement yields overall improvement in wirelength.

Design	Regular Instance		Diced Instance	
	Wirelength	CPU	Wirelength	CPU
Test case 1	2333165	75 secs	2225270	8076 secs
Test case 2	5415380	200 secs	5341744	18132 secs

Table 7: Dicing-based experiment: Comparison between an incremental run on a placement instance with large perturbation, and multiple incremental runs that gradually implement the same perturbation on the same instance. Wirelength is measured for the final incremental run and CPU time is the total time for all incremental runs.

Table 6 shows that incremental optimization gives an overall improvement in solution quality regardless of the size of instance perturbations. **The placement tool's incremental capability may be too strong relative to these magnitudes of instance perturbations.**

6 Routing

Our instances for routing domain are placed circuit netlists. We use the Cadence WarpRoute v1.0.22+ router for our full optimizer, and the same WarpRoute in incremental mode for our incremental optimizer. The initial instance is the original placed design netlist, and instance perturbation consists of changes in cell orientations. To obtain instance I_{j+1} from instance I_j, we randomly select p cells and flip them (i.e., mirror them about the y-axis, so that they can remain in the same sites in the same placement row). When a cell is flipped, all routing information for incident nets is removed.

The full WarpRoute is applied on the initial instance I_0 to obtain an initial solution S_0 for the reversal-based experiment. Instance I_0 is then perturbed by flipping p cells to obtain instance I_1, and we apply incremental WarpRoute to I_1 with S_0 as the starting point, yielding S_1. We continue the process until the last solution S_k is obtained. Again, we reverse the instance sequence so that solution S_{k+1} for instance $I_{k+1} = I_{k-1}$ is obtained with S_k as the initial solution.

Again, two independent runs of routing are used for the dicing-based experiment. The first run creates a new instance I'_k from original instance I_0 by making a big change in the placement (p cells flipped at once). Incremental WarpRoute is called to get S'_k. The second run creates a sequence of instances $I_1, I_2, \dots, I_k$ with $I_k = I'_k$. Incremental WarpRoute is called for each instance.

Experimental Results

For our routing experiments, we use the same two design cases shown in Table 5. Perturbation sizes used for routing experiments are 12, 120 and 1200. We use the same value of $k = 10$ that yields a sequence of 21 routing solutions. Experimental results for all test cases are shown in Table 8. For the dicing-based experiment, we also use an instance that has perturbation size of $p = 1200$ cells. Just as in the placement domain, the first run flips all 1200 cells at once and makes a single call to incremental WarpRoute, while the second run makes 100 small incremental WarpRoute calls with 12 cells flipped between each call. The comparison between two runs is given in Table 9.

Figure 2 shows the wirelength difference between a small perturbation with many runs and a large perturbation with one run. It shows that when performing an incremental updates on the routing instance, it is probably better to perform the update all at once.

From this experiment we can see that in the routing domain, **the perturbation size should be sufficiently large before we apply incremental optimization.** Dicing a large perturbation into smaller ones will degrade the overall solution.

7 Discussion and Conclusions

In the partitioning domain, we can see that applying instance perturbation will yield a better solution. Table 3 shows that for all perturba-

Design		Initial	$p = 12$	$p = 120$	$p = 1200$
Test case 1	Wirelength	3431474	3454783	3515784	3536563
	# Vias	86031	86951	90821	95328
	CPU (secs)	209	1812	2231	4199
Test case 2	Wirelength	7892203	8046985	8471463	8592198
	# Vias	174476	177637	193373	207891
	CPU (secs)	606	3448	5402	13848

Table 8: Routing results for all test cases. All runs are performed on a 300MHz Sun Ultra-10 machine. Wirelengths for incremental runs are the final routed wirelengths (except the initial instance) and CPU times are the total incremental runtime.

Design	Regular Instance		Diced Instance	
	Wirelength	CPU	Wirelength	CPU
Test case 1	3469250	204 secs	3537667	8924 secs
Test case 2	8177713	587 secs	8620049	18442 secs

Table 9: Dicing-based experiment: Comparison between an incremental run on a routing instance with large perturbation, and multiple incremental runs that gradually implement the same perturbation on the same routing instance.

tion sizes, a better result is obtained compared to the initial solution. This also shows that the problem-space metaheuristic approach may be very effective for partitioning – in fact, more effective than realized by previous investigators who studied problem-space methods for partitioning [7]. The authors of [7], as well as the "stable net transition" technique of [16], both use edge deletion (followed by restoration) as the instance perturbation (or, "kick move" in the so-called "large-step Markov chain"[4] approach). In contrast, we use edge reweighting as the instance perturbation, and we automatically achieve the restoration as a consequence of how we reverse the sequence of perturbations to return to the original instance.

In the placement domain, we can see from Table 6 that incremental runs always yield a better solution. This may be because the incremental algorithm for the placement is too strong. In addition, the problem instances we chose may be too easy for the placer, allowing the tool to find a better solution easily.

In the routing domain, we can see from Table 8 that solutions obtained from incremental runs are always worse than the initial solution. One of the reasons for this is that when we remove routing information for some nets, the blockage effect caused by these nets remains intact. In other words, if a net is routed in a certain way such that it blocks several other nets and forces those nets to have detours, then when the routing for this net is removed, those detours will still exist in the overall solution. However, when we try to re-route this specific net within such a context, we may encounter difficulties because most of the spaces have been taken. Thus, interestingly, to achieve convergence in incremental routing, perturbation sizes should be as large as possible before the incremental optimizer is applied.

[4]The Large-Step Markov Chain (LSMC) method of [11] iteratively performs a *descent* using a greedy search engine, and then perturbs the resulting local optimum via a "kick move" to obtain the starting solution for the next greedy descent.

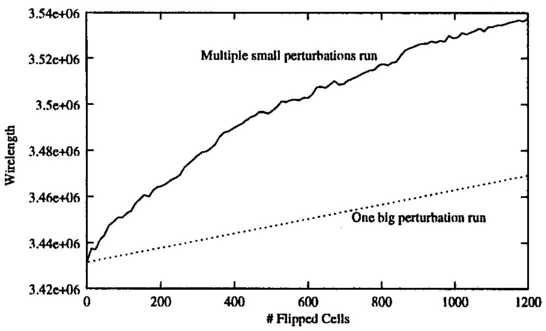

Figure 2: Difference between one run with a big instance perturbation and multiple runs with small instance perturbations. Performing incremental optimization with too-small perturbation sizes will lead to worse solutions.

Altogether, we believe that our experimental results show that current design tools may not be correctly architected to handle incremental optimizations. Because incrementality is such an important aspect of today's deep-submicron design methodologies, it is important for tools to deliver appropriate incremental optimization capabilities. In addition, as overall design *processes* are instrumented to achieve continuous process improvement [6], it is important to understand the ability of tools to improve designs via incremental optimization. By understanding the potential of mismatch between instance perturbation and algorithm strength, designers can find the flow tunings that will best reduce design cycle times and increase designer productivity.

References

[1] C. J. Alpert, "Partitioning Benchmarks for VLSI CAD Community", Web page, `http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html`

[2] C. J. Alpert, "The ISPD-98 Circuit Benchmark Suite", *Proc. ACM/IEEE Intl. Symposium on Physical Design*, April 98, pp. 80-85. See errata at `http://vlsicad.cs.ucla.edu/~cheese/errata.html`

[3] K. D. Boese, *Models for Iterative Global Optimization*, Ph.D. Thesis, UCLA Computer Science Dept., 1996.

[4] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning", *Proc. Asia and South Pacific Design Automation Conf.*, Jan. 2000, pp. 661-666, available at `http://vlsicad.cs.ucla.edu/GSRC/bookshelf`

[5] J. Cong and M. Sarrafzadeh, "Incremental Physical Design", *Proc. ISPD*, 2000, pp. 84-92.

[6] S. Fenstermaker, D. George, A. B. Kahng, S. Mantik and B. Thielges, "METRICS: A System Architecture for Design Process Optimization", *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 705-710.

[7] A. S. Fukunaga, J. H. Huang and A. B. Kahng, "On Clustered Kick Moves For Iterated-Descent Netlist Partitioning", *Proc. IEEE Intl. Symp. on Circuits and Systems*, May 1996, pp. IV/496-499.

[8] B. Hajek, "Cooling Schedules for Optimal Annealing", *Mathematics of Operations Research* 13(2) (1988), pp. 311-329.

[9] D. J. Hathaway, R. R. Habra, E. C. Schanzenbach and S. J. Rothman, "Circuit Placement, Chip Optimization, and Wire Routing for IBM IC Technology", *J. VLSI Signal Processing Systems for Signal, Image, and Video Technology* 16(2-3) (1997), pp. 191-198.

[10] L. N. Kannan, P. R. Suaris and H.-G. Fang, "A Methodology and Algorithms for Post-Placement Delay Optimization", *Proc. ACM/IEEE Design Automation Conf.*, 1994, pp. 327-332.

[11] O. C. Martin, S. W. Otto and E. W. Felten, "Large-Step Markov Chains for the Traveling Salesman Problem", *Complex Systems*, 5(3), 1991, pp. 299-326.

[12] I. H. Osman and J. P. Kelly, eds., *Meta-Heuristics: Theory and Applications*, Kluwer, 1996.

[13] R. Otten, "Global Wires Harmful?", *Proc. Intl. Symposium on Physical Design*, 1998, pp. 104-109.

[14] R. Otten and R. K. Brayton, "Planning for Performance: the Constant Delay Paradigm", *Proc. ACM/IEEE Design Automation Conf.*, 1998.

[15] J. C. Shah and S. S. Sapatnekar, "Wiresizing With Buffer Placement and Sizing for Power-Delay Tradeoffs", *Proc. Intl. Conf. on VLSI Design*, Bangalore, 1996, pp. 346-351.

[16] T. Shibuya, I. Nitta and K. Kawamura, "SMINCUT: VLSI Placement Tool Using Min-Cut", *Fujitsu Scientific and Technical Journal* 31(2) (1995), pp. 197-207.

[17] Semiconductor Industry Association, "The National Technology Roadmap for Semiconductors: Technology Needs", December 1997.

[18] R. H. Storer, S. D. Wu and R. Vaccari, "New Search Spaces for Sequencing Problems With Application to Job Shop Scheduling", *Management Science* 38 (1992), pp. 1495-1509.

[19] W. Sun and C. Sechen, "Efficient and Effective Placements for Very Large Circuits", *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, 1993, pp. 170-177.

[20] M. Wang, P. Banerjee and M. Sarrafzadeh, "Potential-NRG: Placement With Incomplete Data", *Proc. ACM/IEEE Design Automation Conf.*, 1998.