An Incremental Floorplanner[1]

Jim Crenshaw
Corporate Research Labs
Motorola
Ajc030@email.mot.com

Majid Sarrafzadeh and
Prithviraj Banerjee
ECE Department
Northwestern University
{majid, banerjee}@ece.nwu.edu

Pradeep Prabhakaran
Compaq-Digital
pradeep@galois.hlo.dec.com

Abstract

One of the foremost problems in physical design for deep-submicron circuits is the need for estimates that depend on future decisions. Estimation of area, timing, and coupling are required..We propose a novel floorplanner, with a new wiring metric, which can be updated quickly in small increments. This provides tools with a way to influence the floorplan as they make changes without a large running time penalty. We provide experimental results that show the incremental approach to be generally 5 times faster than full floorplanning while maintaining good estimates.

1 Introduction

Many of the goals of design automation require information about decisions that will be made later in the design process. For instance, the longest-delay path used in the circuit determines the clock cycle period. However, the delay in a path is dependent upon the capacitance of the nets in the path, which isn't fixed until routing has been done on the design. On the other hand, some physical parameters can be approximated at a higher level. For instance area can be approximated once an allocation has been decided. In the case of net capacitance, which is based on wirelength, an estimate is easily obtained after floorplanning by taking the perimeter of the smallest bounding box enclosing the modules to which the net is connected.

The problem with using traditional floorplanners is that they are too slow to invoke every time a potential move in the design space is to be evaluated. To address this, we studied ways to update the floorplan incrementally since typical design space exploration involves small changes that need not impact the layout. We show that the incremental approach to floorplanning can be fast and still supply other tools with good physical estimates for area and wirelength.

2 Previous Work

The need to update the floorplan as decisions are made has not been studied as such, although some attempts have been made to combine some algorithms directly with floorplanners.

The idea of using a slicing tree for floorplanning was introduced in [1], and has been included in many textbooks (for instance, [2]). Research which combines high level synthesis and floorplanning has been described in [3], [4], [5]. In [3], a full floorplan is done for each high level move, which is much slower than our incremental approach. Our work contrasts with that of [4] because they use a binding algorithm for each floorplan move. The work described in [5] specifies an algorithm which does floorplanning locally for each path in the data-flow graph (simultanesouly scheduling and binding those paths), and then attempts to improve the bindings on the result. Thus, their approach is not incremental and not easily adapted to other algorithms.

High level synthesis efforts have always included area and wiring estimates. Traditional methods such as those found in [6-9] have typically relied on library cell area measurements and gate capacitance. While this was appropriate before deep submicron design became popular, wirelength must be included in any estimate today. Other research such as [10] uses statistical analysis of previous designs done with the same tool flow. The problem with this approach is sensitivity to tool changes. Some have also required wire capacitance not only for delay, but also

[1] This work supported in part by NSF Grant number MIP 9527389; a contract by DARPA (ACS) and Motorola Center for Communications.

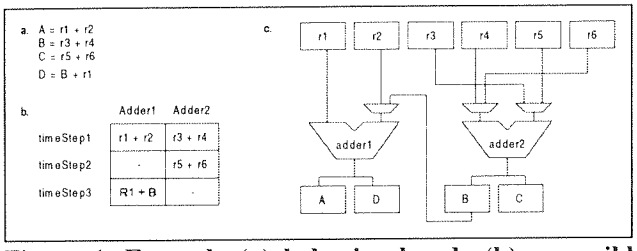

Figure 1. Example (a) behavioral code (b) a possible schedule and (c) a possible datapath.

for power estimation, which is a function of switched capacitance [11, 12]. In the former method, gate capacitance was again used, and in the latter, no explicit capacitance metric was specified.

3 Incremental Floorplanning

In this section, we present an intuitive discussion of the incremental floorplanning problem as applied to high level synthesis, and then we present some formal results that hold for the general incremental floorplanning problem. We propose an approach that supposes a fixed allocation, and changes the timestep during which an operation is executed, or changes the functional unit to which an operation is bound.

To illustrate a schedule change and the resulting impact on a floorplan, consider Figure 1. In part **a**, we have a small piece of behavioral code, which performs four additions. Part **b,** shows a possible schedule and binding for the code, and **c** shows a datapath capable of executing the code. Figure 2 shows a possible floorplan. Now, a change in only the schedule amounts to changing the timestep in which an operation is

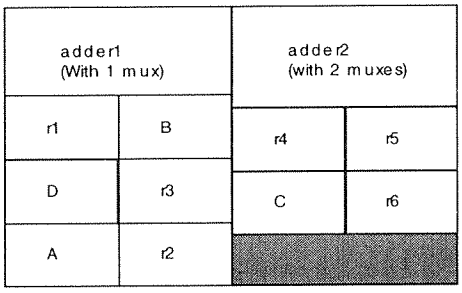

Figure 2. Possible floorplan for the netlist in Figure 1.

done, without changing the binding. For instance, suppose that r1+r2 is rescheduled for the second timestep. Note that the original datapath can still calculate the code. Therefore, the floorplan has not changed, and so no floorplan update is needed for this high level move. This is true in general for scheduling only changes.

Consider the case of a binding change. Suppose that a move is made from the schedule and binding in Figure 1a, so that *r3 + r4* is done on *Adder1* instead of *Adder2*. The resulting schedule is shown in Figure 3a. Note that to make this change, *Adder1* must now have a path from *r3* and *r4*, which were not present in the original datapath. Also new multiplexors must be added. The resulting datapath is shown in Figure 3b. Clearly this may cause some changes to the floorplan. Figure 2 shows a possible slicing floorplan for the example of Figure 1.

Note that multiplexors are combined with the functional units to which their outputs are connected. Now, consider the same floorplan after the binding change of Figure 3 is made. The result can be seen in Figure 4. Clearly the result is larger, and it is easy to see how it might be improved (for instance by moving registers *A* and *r2* to the right).

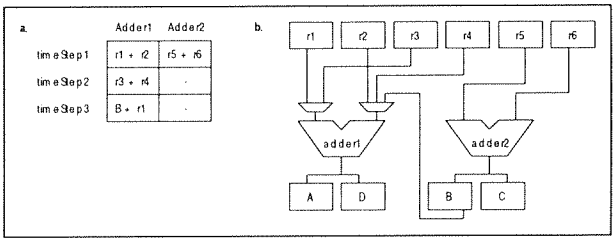

Figure 3. Example of results from binding change.

Lemma: A single wiring change to an optimal floorplan may require Omega(n) cells to be moved in order to maintain optimality. Proof omitted.

This shows that the asymptotic worst case running times of the incremental floorplanner is as bad as regular floorplanning. To address this, we allow some error to develop which results in implicit overlap of modules during floorplanning. It is clear from the previous discussion that best case runtime is considerably better for the incremental method. We

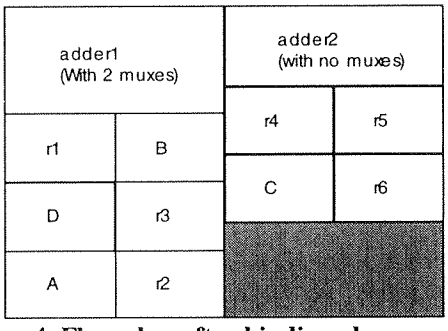

Figure 4. Floorplan after binding change.

will show that in practice, the average run time is also much better.

4 Proposed Algorithm

We propose an algorithm with three phases. In the initial phase, a full floorplan is created from a baseline netlist supplied by the synthesis tool. During the next phase, as new design automation decisions

are evaluated, updates are made incrementally on the floorplan. In the final phase, the floorplanner is run one last time on the entire netlist. This last phase can be used as an indication of whether the incremental updates were sufficient (if it is a significantly better solution, then a better update threshold should be used).

The cost function driving the floorplanner is based on minimizing area, and it incorporates a new wire area metric. Since deep-submicron designs have the potential for higher wire area to gate area ratios, it is important to account for the change in area due to wiring. Each wire's contribution to area is derived as follows. The bounding box of the wire is computed, and each module which is intersected by the wire adds one wire-unit length to both the x and y dimension.

The initial floorplan is done by partitioning the netlist based on a minimum net cut metric. The Fidducia-Matheyses Algorithm is used to do this because of its low time complexity. The partition is required to meet a balance criterion based on area, which is computed as the sum of module areas. At this stage, module implementation selection has not been done, so a library supplied area index is used for each module type. Partitioning is applied repeatedly until only single modules are left. The resulting binary tree structure of netlists is then used to build a slicing tree as follows. Using library defined implementation lists for each module added to a muxltiplexor estimate, a stairstep curve is created bottom up for each node in the tree. At the top level, the best solution is selected, and this selection is propagated down the tree to the module level. Finally, based on the initial floorplan, wire estimates are added and the plan is updated to reflect the wiring. These estimates are calculated by finding the bounding box for each net, and adding one wire unit to each module intersecting it.

During the update phase, for each high level move, the affected nets are first removed from the wirelength calculation, updated to reflect new pins or deleted pins, and then put back into the wirelength. Multiplexing is similarly updated for modules with input changes (if any). Next, any modules that changed greater than a specified threshold since the last time they were updated are marked critical, and their parent nodes in the slicing tree are put into the critical queue. While the queue is not empty, a node is removed and checked to see if it is still critical. If so, the floorplanning algorithm from phase one is reapplied to it (and recursed down). All of its subnodes are marked non-critical. If the resulting implementation list does not contatain a solution within threshold, the node's parent is added to the critical list. Once the critical queue is empty, the update is complete. Although in the worst case this method does twice as much work as a full floorplan, we will see that it rarely has to work that hard.

Note that the threshold may be two-sided so that not only larger area changes require updating, but also those which are much smaller than the previous space. This may indicate an opportunity for the floorplanner to improve the global solution. Pseudocode for the primary method, which performs the incremental updates, is shown in Figure 5.

In the pseudocode, several methods are mentioned. RemoveAffectedNets() strips the nets which will change from the floorplan, then they are changed as specified by updateAffectedNets(), and finally replaceAffectedNets puts them back into the floorplan, and any leaf cells which are now over threshold are placed in a queue and returned. The question of whether a particular node is over threshold is answered by the boolean method stillCritical(). If a node removed from the queue is critical, its sub-floorplan is recomputed, and if the result is over threshold, its parent is entered into the queue. If the sub-floorplan fits, then all of its descendants are marked non-critical in case any of them are still in the queue. Once the queue is empty, either all nodes are now under threshold. In the worst case, the top level is re-floorplanned and all thresholds are reset.

Lemma. Worst case time complexity is the same as partitioning based floorplanning, $O(n^2 + pn)$, where n is the number of modules, and p is the number of pins.
Proof omitted for brevity.

```
Method incrementalUpdate(NetChange change) {
    removeAffectedNets(change);
    updateAffectedNets(change);
    criticalQueue = replaceAffectedNets(change);
    while (criticalQueue.notEmpty()) {
        critNode = criticalQueue.next();
        if (critNode.stillCritical()) {
            critNode.floorplanSubtree();
            if (critNode.stillCritical()) {
                Queue.add(critNode.returnParentNode());
            } else {
                for each descendant, d, of critNode {
                    d.setNotCritical();
                }
            }
        }
    }
}
```

Figure 5. Pseudocode for incremental update method.

5 Experimental Data

We implemented our algorithm to perform high level synthesis on three different algorithms, with four different allocations, thus generating twelve different netlists. Each netlist was captured in its initial state, along with the list of incremental changes resulting in its final state. These changes were then applied using the incremental floorplanner to evaluate its performance.

Table 1. Partition count ratio.

	Vec3	Vec4	Vec5	Vec6
Bres	3	6	8	11
Heap	443	9	392	501
Clip	49	15	15	19

The speedup of the incremental approach is illustrated in Table 1. Rather than using time, we measure the number of nodes partitioned over the entire file of incremental changes. This is a better metric because is closely related to the smallest operations being done by the program, whereas time is influenced by the operating system. Here we can see that in all tested cases, the incremental method was usually at least five times as fast, and was often two orders of magnitude faster. The same data is shown graphically in Figure 6. Note that although the netlist in EX50 appears to degrade with more moves, it is actually generating better floorplan solutions for the extra effort.

Another important issue regarding the use of the incremental technique is whether it results in conservative estimates for physical parameters such as wirelength. In Table 2, the wirelength percent difference is shown. Each cell is calculated as the difference of incremental wirelength estimate minus full floorplan wirelength over full floorplan wirelength. Note that in the case of Bres, which had the worst time improvements over full floorplanning, the algorithm is able to find floorplans with significantly improved wirelength. This is due to the use of local action, which may make better decisions that the top down full floorplanner may disallow as a result of a poor early decision.

Table 2. Total wirelength % difference.

	Vec3	Vec4	Vec5	Vec6
Bres	28	20	17	20
Heap	-12	-1	-11	-10
Clip	-19	-29	-5	-11

6 Conclusion

We have presented an algorithm which incrementally updates a slicing floorplan, and we have shown experimentally that it is fast and produces good results.

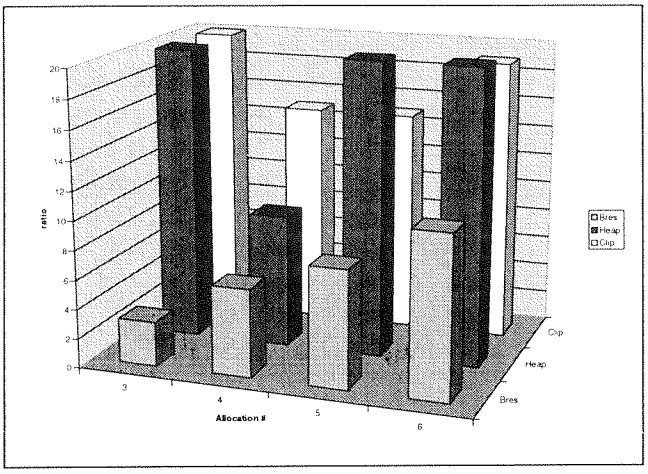

Figure 6. Partition Ratio plot. Each row is a single algorithm synthesized over four different allocations. The resulting ratio of nodes partitioned in full floorplanning to incremental varies from 2 to over 20.

7 Bibliography

[1] D. F. Wong and C. L. Liu, "A New Algorithm for Flooplan Design," presented at 23rd Design Automation Conference, 1986.

[2] M. Sarrafzadeh and C. K. Wong, *An Introduction to Physical Design*: McGraw Hill, 1996.

[3] P. Prabhakaran and P. Banerjee, "Simultaneous Scheduling, Binding and Floorplanning in High-Level Synthesis," presented at VLSI Design, Chennai, India, 1998.

[4] Y. Fang and D. F. Wong, "Simultaneous Functional Unit Binding and Floorplanning," presented at ICCAD, 1994.

[5] J. Weng and A. C. Parker, "3D Scheduling: High-Level Synthesis with Floorplanning," presented at 28th Design Automation Conference, 1991.

[6] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis*. Boston: Kluwer, 1992.

[7] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 85-93, 1991.

[8] W. Wolfe, A. Takach, C.-Y. Huang, R. Manno, and E. Wu, "The Princeton University Behavioral Synthesis System," presented at 29th ACM/IEEE Design Automation Conference, Anaheim, California, 1992.

[9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*: McGraw-Hill, 1994.

[10] P. Landman and J. Rabaey, "Architectural Power Analysis: The Dual Bit Type Method," *IEEE Trans. on VLSI Systems*, vol. 3, pp. 173-187, 1995.

[11] A. Raghunathan and N. Jha, "An Iterative Improvement Algorithm for Low Power Data Path Synthesis," presented at International Conference on Computer Aided Design, 1995.

[12] J. Crenshaw and M. Sarrafzadeh, "Accurate High Level Datapath Power Estimation," presented at European Design and Test Conference, 1997.