Optimal Latch Mapping and Retiming within a Tree

Joel Grodstein, Eric Lehman, Heather Harkness, Herve Touati, and Bill Grundmann

Digital Equipment Corporation, Hudson, MA

Abstract. We propose a technology mapping algorithm that takes existing structural technology-mapping algorithms based on dynamic programming [1,3,4] and extends them to retime pipelined circuits. If the circuit to be mapped has a tree structure, our algorithm generates an optimal solution compatible with that structure. The algorithm takes into account gate delays and capacitive loads as latches are moved across the logic. It also supports latches with embedded logic: i.e., cells that combine a D latch with a combinational gate at little extra cost in latch delay.

1. Introduction. The problem of retiming for minimal area under delay constraints using transparent latches and reasonably accurate delay models is still unsolved. Leiserson and Saxe[5] developed an optimal solution for edge-triggered D flip-flops, under the assumption that capacitive loads and gate delays are not modified when latches are moved. Ishii et. al. [6] gave an optimal algorithm that also handles transparent D latches. Locklear et. al. [7] extended this solution to take clock skew into account.

All the above algorithms are *globally* optimal; but they achieve this by compromising their local accuracy. None model load-dependent delays; moving latches changes node loads and thus gate delays. Furthermore, all are restricted to the use of D latches. Many cell libraries contain not only D latches but also latches with embedded logic. Such latches, whose function is of the form **Q=latch(clk,f(x1,x2,..,xn))** combine a simple transparent latch and a gate (denoted **f(x1,x2,...,xn)**) for only little more delay and area than the latch itself. These cell libraries are often incomplete; e.g., a typical cell library based on tristate latches may embed only D, AND2, AND3, and OR2 gates in latches, disallowing larger gates and inverting gates. For such asymmetric libraries, a simple change of latch polarity (i.e., retiming across an inverter) may enable or disable the embedding of a gate within the latch. Such effects are not modeled by previous work.

Furthermore, the application of existing retiming algorithms [5,6,7] *after* technology mapping is suboptimal. The smallest possible latch movement is a movement across a single mapped gate. But this ignores the possibility of breaking a large gate into smaller pieces and placing the latch between them to meet a delay constraint.

As a consequence, existing global retiming algorithms cannot fine-tune latch placements. By contrast, our algorithm performs retiming during technology mapping, which provides more control on latch placement and leads to a more efficient use of latch-embedded logic.

Tree mapping [1,3,4] provides technology-dependent local optimization in linear-time. For a given cell library and initial tree decomposition of a circuit, tree mapping is optimal. However, standard tree mapping does not handle latches; they are considered as tree roots only [9].

We extend tree mapping to networks that are trees, but can contain latches at interior points. We allow these latches to be moved anywhere *within* a tree, but not *across* trees. We preserve the tree mapping algorithm's optimality properties and its linear-time complexity, and handle both edge-triggered and level-sensitive latches. This has several gains:

- Critical paths are reduced not only by moving latches but also by embedding logic within latches.
- Areas and loads are correctly modeled as latches move within a tree.
- We retime at the level of a tree-matching primitive (a 2-input NAND gate or an inverter), which is finer than the gate level.

The rest of the paper is organized as follows. In Section 2 we extend tree mapping to allow latches and retiming within trees. In Section 3, we deal with level-sensitive and conditional latches. We give results and conclusions in Section 4.

2. Latch mapping and retiming algorithm.

For ease of illustration, we describe our retiming algorithm in terms of a min-area tree mapper [4]. It can easily be adapted to minimize delay [1] or area under delay constraints [2].

The algorithm starts with a decomposition of the subject tree into 2-input NANDs and inverters [3,4]. Any node in the tree can be marked as a latching point. We have a standard target library of combinational gates, predecomposed into two-input NAND gates and inverters. We have an additional library of latches, of the form **Q=latch(ϕ, F(x1,x2,...xn))**. This library is pre-decomposed based on F, *exactly as if there were no latching involved.* We then implement basic latch mapping simply: if we wish to place a latch on a node, we merely restrict ourselves to using matches from the latch library instead of the gate library, and connect its clock to the appropriate global clock. The following algorithm provides retiming:

```
set all solutions to high-cost;
for each node N in the tree, inputs to outputs
{
    if N is a leaf
        N.sol[weight(N)] = 0; continue;
    for each match M from the comb. library
        for w=0..MAX_WEIGHT
            N.sol[w] = MIN (itself,
                        calculate_match(M,w));
    for each match M from the latch library
        for w=0..MAX_WEIGHT
            N.sol[w] = MIN(itself,
                    calculate_match(M,w+1));
}
calculate_match (match, weight)
{
    area = area of the gate from match;
    for each input pin IP of match
        area += area of IP.sol[weight];
    return (area);
}
```

We start by tabulating, at each leaf node **n**, the number of latches weight(**n**) between **n** and the tree root. Assume for now that we use edge-triggered flip-flops. Then any retiming which preserves weight(**n**) for each leaf **n** is legal. We next extend our data structure. Instead of having one solution per node, we store a solution for each retiming weight per node. A node implementation has a retiming weight of **w** if, under this implementation, there must be exactly **w** latches between the output of the node and the root of the tree. At any node, only one of these solutions will be used in the final mapping. For each leaf **n,** we initialize **n**.sol[weight(**n**)] with zero area. We then perform tree mapping. For each internal tree node I, we compute the best solution for each retiming weight. For a given weight **w** we can:

1. implement I with a gate; at each gate input, use input.sol[**w**] as the input cost.
2. implement I with a latch; use input.sol[**w**+1] as the input pin cost at each latch input.

When we reach the root of the tree, the optimal implementation is in root.sol[0].

Consider the NAND-inverter tree in Figure 1, with a single phase-1 latch at G. There is one latch between each of I1,I2 and the tree root D. Thus any solution for E.sol[0] needs to contain a latch on each path between I1 or I2 and E. For example, E.sol[0] could consist of a phase-1 AND2 latch driven by I1.sol[1] and I2.sol[1] (Figure 2). Any tree implementation that uses E.sol[0] will not contain any latches downstream of E.

At node F, we could store in F.sol[0] an implementation made of E.sol[0] followed by an inverter (Figure 3).

At node G, we could implement G.sol[0] with a NOR2 gate and drive the gate with weight 0 inputs (Figure 4). By using weight 0 inputs on F and J, we assume that latches are placed before F and J. Alternatively, we could implement G with a latch, driven by weight-1 implementations (Figure 5). In this example the algorithm would examine both solutions and select the one with minimal area.

3. Complex latches. Edge-triggered latches are the simplest case. The algorithm can also handle level-sensitive and conditional latches. We must now preserve not only the *number* of latches between the inputs and the root, but also the correct *sequence*; this avoids arbitrary swappings of latch positions. In this case, the simple array of solutions is replaced by an array where each entry represents a prefix of the correct sequence of latches. Furthermore, simple flow-through latch calculations must be performed at each latch.

Cells with reconvergent embedded logic (e.g., multiplexors or JK flops) can only be used by a tree mapper at tree boundaries. Our algorithm retains this limitation. However, if a tree has a mux at its leaves, we can retime a downstream latch back toward the leaves to use a mux-embedded latch, no matter where the latching point originally was.

4. Results. We have incorporated these algorithms into a new technology mapper, SynFul, built over SIS [8]. We map for minimal area under a delay constraint using an algorithm based on [1,2]. Our library has transparent non-inverting latches only, with D latches, embedded AND2, AND3, and OR2 latches. We have taken several large examples and mapped them first with no retiming (column marked *orig*inal). Then, we have remapped them allowing retiming only across a single inverter. Finally, we have allowed arbitrary retimings. A * indicates that delay constraints were not met. The net results are geometric means of the improvements relative to the original unretimed mapping. Execution times were insignificant; tree mapping is a linear-time algorithm.

	# gate	# lat	orig area	INV area	ret. area	ret. #lat
e_lu0	83	26	33*	322	29	20
c_baf	77	22	25*	23	21	19
ipe1c	121	67	239	213	196	65
ipe2c	102	65	197	180	141	43
aadc1	83	34	124	118	121	38
e_shf	25	6	344	298	303	7
immc	64	24	122	113	114	26
NET		1.0	1.0	.91	.84	.92

Allowing retiming across an inverter improves area in every example (an average of 9%) while also eliminating delay-constraint

violations. Allowing arbitrary retimings reduces area an additional 7%. Furthermore, by pushing latches forward when possible, it reduces latch count by 8%.

Note that SynFul performs area recovery during mapping; i.e., the added slack introduced by latch movement is heuristically used to reclaim combinational area. The non-optimality of this heuristic explains the few cases where arbitrary retiming yields increased area.

In conclusion, we have improved tree mapping in several ways. By allowing latching points in the interior of trees, we allow trees to span several phases and thus to be larger. Furthermore, we allow the technology mapper to perform retiming within a tree. This retiming is optimal for a tree, for a given initial decomposition of a network and ignoring reconvergence at tree leaf nodes. Finally, for many libraries, we obtain significant advantage using latches with a logic function embedded within them.

References:

1. Touati, H., "Performance-Oriented Tech. Mapping," PhD Thesis, UCB/ERL M90/109, U.C. Berkeley 1990.
2. K.Chaudhary et. al., "A Near-Optimal Algorithm for Technology Mapping Minimizing Area under Delay Constraints," DAC-92.
3. E. Detjens et. al., "Technology Mapping in MIS," 1987 ICCAD, pp. 116-119.
4. R. Rudell, "Logic Synthesis for VLSI Design," Ph.D. thesis, UCB/ERL M89/49, U.C. Berkeley, 1989.
5. Leiserson and Saxe, "Retiming Synchronous Circuitry," Algorithmica, 6(1) 1991.
6. Ishii,et al, "Optimizing Two-Phase, Level-clocked Circuitry," in Adv. Research in VLSI: Proc 1992 Brown/MIT Conference.
7. Locklear,B, et al, "The Practical Application of Retiming to High-Perf. Systems," ICCAD 1993.
8. Sentovich et al "SIS: A System for Sequential Circuit Synthesis," ICCD-92, pp.328.
9. C. Moon et al, "Technology Mapping for Sequential Logic Synthesis," Proc. Intl. Workshop on Logic Synthesis, North Carolina, 1989.

Figure 1

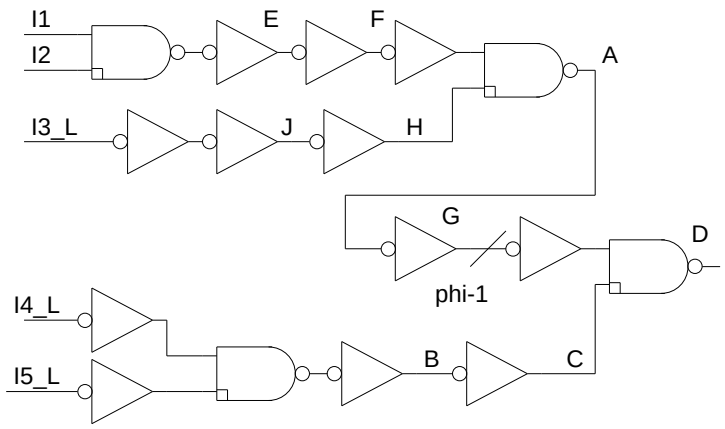
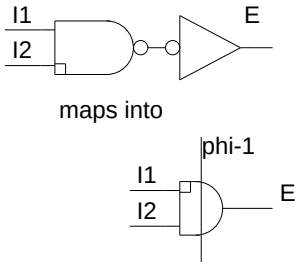
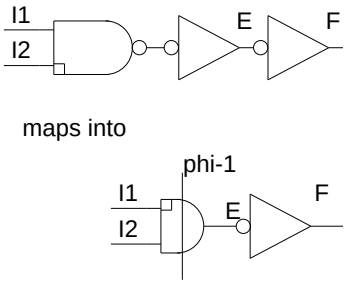


Figure 2



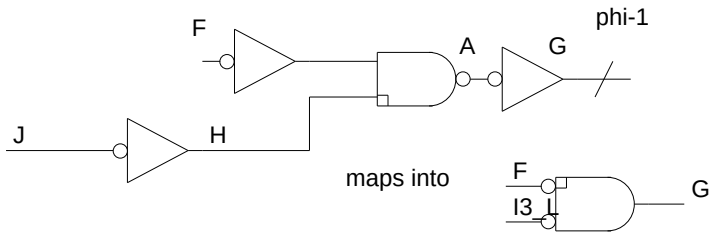
E.sol[0] = LATCH
(I1.sol[1], I2.sol[1])

Figure 3



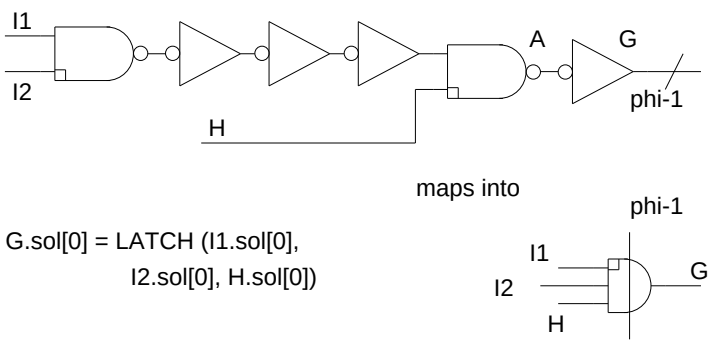
F.sol[0] = INV (E.sol[0])

Figure 4



G.sol[0] = NOR2 (F.sol[0], J.sol[0])

Figure 5



G.sol[0] = LATCH (I1.sol[0],
I2.sol[0], H.sol[0])