Power-Switch Routing for Coarse-Grain MTCMOS Technologies

Tsun-Ming Tseng, Mango C.-T. Chao
Dept. of EE, National Chiao-Tung University
Hsinchu, Taiwan
strayleaf.ee95@nctu.edu.tw
mango@faculty.nctu.edu.tw

Chien-Pang Lu, Chen-Hsing Lo
Mstar Semiconductor
ChuPei, Taiwan
knuth.lu@mstarsemi.com
jack.lo@mstarsemi.com

ABSTRACT

Multi-threshold CMOS (MTCMOS) is an effective power-gating technique to reduce IC's leakage power consumption by turning off idle devices with MTCMOS switches. However, few existing literatures have discussed the algorithms required in MTCMOS's back-end tools. In this paper, we propose a switch-routing framework which serially connects the MTCMOS switches without violating the Manhattan-distance constraint. The proposed switch-routing framework can simultaneously maximize the number of MTCMOS switches covered by its trunk path and minimize the total path length. The experimental result based on four industrial MTCMOS designs demonstrates the effectiveness and efficiency of the proposed framework compared to a solution provided by an EDA vendor and an advanced TSP solver.

1. INTRODUCTION

By using both high-V_t and low-V_t transistors, *Multi-threshold CMOS* (MTCMOS) emerges to be an effective power-gating technique which can simultaneously reduce leakage power and maintain circuit performance. For an MTCMOS design, the header/footer switches and retention flip-flips are implemented in high-V_t transistors, such that their leakage current can be lowered during the sleep mode. On the other hand, the power-gated logics are implemented in low-V_t transistors, such that their performance can be increased during the active mode. The MTCMOS designs can be classified into two categories by its granularity: (1) *fine-grain* MTCMOS, in which one switch is built into each cell, and (2) *coarse-grain* MTCMOS, in which one switch is built to turn off a block of cells. In this paper, we only focus on the discussion of the coarse-grain MTCMOS.

A lot research effort has been put into the area of MTCMOS technologies during the past decade. One group of the research works focus on the MTCMOS power-gating structures, such as (1) the charge-recycling technique [1] [2], which can reduce the power consumption produced during the sleep-to-active mode transition, (2) the intermediate-mode switches [3], which can cover various demands of the power-performance trade-off, and (3) the distributed sleep-transistor network [4], which can effectively reduce the area overhead of sleep transistors. Another research aspect is to optimize different parameters under different constraints, such as circuit performance, mode-transition power consumption, power-supply noise, area overhead, and wake-up time, by using the sleep-transistor sizing [5] [6], circuit clustering [7], wake-up scheduling [8] [9], or simultaneous clustering and scheduling [10]. However, by our best knowledge, no previous work has discussed the required back-end algorithms supporting MTCMOS technologies.

A back-end tool for coarse-grain MTCMOS technologies should provide the following two functionalities: the *switch allocation* and the *switch routing*. In the MTCMOS design flow, designers first determine the area ratio of MTCMOS switches over the total cells based on the worst IR drop which can be tolerated on the power rails. A higher MTCMOS-switch ratio leads to a lower IR drop but, at the same time, a higher area overhead. Next, the switch allocation can determine the placement pattern and the spacing between adjacent MTCMOS switches based on the above area ratio of MTCMOS switches. Using the obtained placement pattern and spacing, the switch allocation then evenly distributes the MTCMOS switches over the entire IC except its hard macros, where no MTCMOS switches can be inserted. Thus, IC's floorplan should be finalized before distributing MTCMOS switches, but the placement of the gated low-V_t cells is not done yet.

After the location of each MTCMOS switch is determined, the switch routing will serially connect the sleep/wake-up signal of MTCMOS switches one by one. This serial connection can be viewed as a Hamiltonian path of MTCMOS switches. The main reason of using serial connection of MTCMOS switches instead of parallel connection is to reduce the rush current produced during sleep/active mode transition. Also, the signal at the end of the Hamiltonian path can be used as an acknowledgement signal, showing that all the MTCMOS switches are successfully turned on. Besides finding a feasible Hamiltonian path of MTCMOS switches, the switch routing also attempts to minimize the length of the Hamiltonian path so that more routing resources can be left for the routing of low-V_t cells. However, not any pair of switches can be connected to each other. The Manhattan distance between two connected switches has to be under a limit. Otherwise the loading of a switch may

ICCAD 2009, DOI: 10.1145/1687399.1687408
http://ieeexplore.ieee.org/document/5361318/
http://dl.acm.org/citation.cfm?id=1687408

violate its loading constraint defined in the timing library, resulting in an unpredictably long signal delay. In addition, the hard macros in the floorplan may break the regularity of switches' placement, which further increases the difficulty of finding a feasible Hamiltonian path of switches.

In this paper, we propose a switch-routing framework for coarse-grain MTCMOS technologies, which first attempts to find a feasible Hamiltonian path covering all MTCMOS switches under the Manhattan-distance constraint and also minimize the length of the Hamiltonian path. If a feasible Hamiltonian path cannot be obtained, an acyclic *trunk path* covering a maximal number of switches will be reported along with *branches* covering the rest switches. The proposed switch-routing framework can not only take advantage of the location regularity of most switches, such that the path length can be minimized, but also provide the flexibility to handle the location irregularity of some switches caused by hard macros, such that the resulting trunk path can cover maximal switches. The proposed switch-routing framework is embedded in a design flow of the MTCMOS technology provided by an IC foundry [11]. Our experimental result on four industrial MTCMOS designs first shows that the proposed framework can effectively find a feasible Hamiltonian path with shorter length and shorter turn-on time compared to a rough solution provided by an EDA vendor. The experimental result also demonstrates that such a feasible Hamiltonian path is difficult to be obtained by a TSP solver. The reported runtime further demonstrates the scalability of the proposed switch-routing framework for industrial MTCMOS designs.

2. BACKGROUND

In this section, we introduce the MTCMOS technology and some related background information used in the proposed switch-routing framework. We focus on the discuss of MTCMOS designs using header switches. But the proposed framework can be easily applied to footer-switch MTCMOS designs in a similar manner.

2.1 Architecture of MTCMOS Designs

Figure 1 first illustrates the overview of an MTCMOS design using header switches. The power-gated low-V_t cells are connected to the virtual V_{DD}, whose power supply is controlled by the MTCMOS switches placed between the virtual V_{DD} and true V_{DD}. When the system turns on the wake-up-request signal, the header switches are turned on in order so that the virtual V_{DD} can obtain the power from the true V_{DD}. After the system receives the wake-up-acknowledge signal, the system starts to send jobs to the gated logics. When the system turns off the wake-up-request signal, the switches are turned off and the virtual V_{DD} cannot provide any power to the gated cells, meaning that no leakage current can be generated on the gated cells.

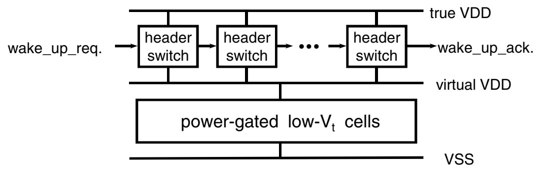

Figure 1: MTCMOS power-supply architecture using header switches

Figure 2 shows the physical layout of this power-supply architecture. On the vertical power meshes, the true V_{DD} and V_{SS} are alternatively provided. On the horizontal power rails, the virtual V_{DD} and V_{SS} are alternately provided, where an MTCMOS switch is placed to control the connection between a true-V_{DD} mesh and a virtual-V_{DD} rail. A V_{SS} rail is directly connected to a V_{SS} mesh through a via. Therefore, the location of an MTCMOS switch must be on the intersection of a vertical power mesh and a horizontal power rail. The spacing between two power rails depends on a standard cell's height and the spacing between two power mesh depends on the whole chip's power planning.

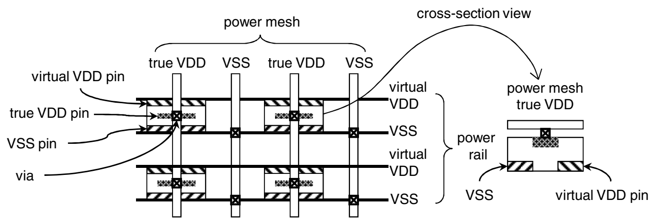

Figure 2: Physical layout of the MTCMOS power-supply architecture using header switches.

2.2 Switch Allocation

The area ratio of the MTCMOS switches over the total cells determines the IR drop on the power rails as well as the placement pattern for the switch allocation. The switch-placement pattern used in our MTCMOS designs is similar to Figure 3.

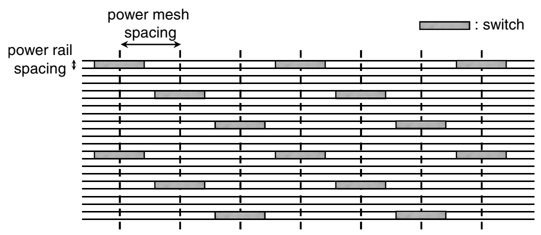

Figure 3: Placement pattern of MTCMOS switches.

According to the switch-placement pattern, the switch allocation then evenly places the MTCMOS switches over the IC except the hard macros, where no switch can be inserted. If the encountered hard macro is an SRAM core or any hard IP using the same power domain as the standard cells, extra switches are placed along the boundaries of the hard macro to strengthen its power supply. If the encountered hard macro is an analog IP, which has its own power domain, the switch-placement pattern around the boundaries remains the same. Figure 4 shows an exemplary switch allocation with hard macros. As a result, the regularity of the switch-placement pattern is broken by the hard macros.

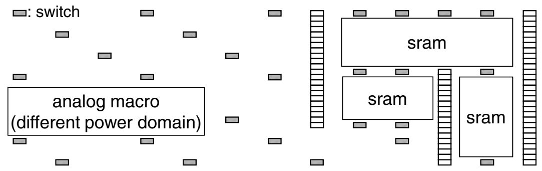

Figure 4: Switch allocation with hard macros.

2.3 MTCMOS Switches and Switch Routing

The coarse-grain library [11] provides two types of header switches: the *single-input switches* and *double-input switches* as showed in Figure 5(a) and 5(b), respectively, where all the inverters in Figure 5 are always-on and get their power directly from the true V_{DD}. If the single-input switches are used, the switches are serially connected as Figure 6(a), where the $NSIn$ signal of the first routed switch is connected to system's wake-up-request signal and the $NSOut$ of the last routed switch is connected to the system's wake-up-acknowledge signal. Since the pins of wake-up-request and wake-up-acknowledge signals usually locate next to each other, the routed path of single-input switches looks like a Hamiltonian cycle.

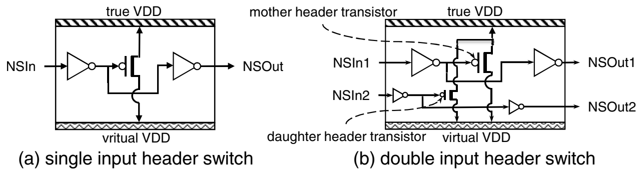

Figure 5: Types of MTCMOS header switches.

If the double-input switches are used, the switches are connected as Figure 6(b), where the wake-up-request and wake-up-acknowledge signals are connected to the $NSIn2$ and $NSOut1$ signals of the first routed switch, respectively. Also, the $NSOut2$ signal of last routed switch is connected to the $NSIn1$ signal of itself. Therefore, the routed path of double-input switches looks like a Hamiltonian path. The routed path can end at any switch in the design.

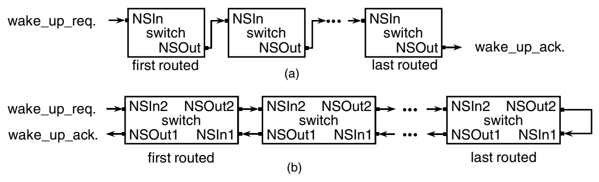

Figure 6: Switch routing for (a) single-input and (b) double-input switches.

The double-input switches can reduce the power-up glitch current by first turning on a daughter header transistor with smaller driving capability and strengthen the power supply to the virtual V_{DD} by then turning on a mother header transistor with larger driving capability. In our MTCMOS designs, we use double-input switches as recommended by [11].

2.4 Manhattan-Distance Constraint

When connecting the next routed switch, designers have to make sure that the output loading of the current switch (or the input slew of the next routed switch) cannot exceed the upper bound of the timing library. Otherwise, the signal delay between two switches may be unpredictably long. One practical solution is to set a constraint on the Manhattan distance between two connected switches based on metal's unit-length loading and switch's intrinsic loading. This Manhattan-distance constraint has to be conservative since the detail routing path between two switches may detour due to routing congestion. In fact, this constraint is usually an empirical value and may vary from different designs and adopted APR tools.

3. PROBLEM FORMULATION OF MTCMOS SWITCH ROUTING

3.1 Problem Formulation

Given the result of the switch allocation, the switch routing is performed to find a routing path which can serially connect all the MTCMOS switches and satisfy the Manhattan-distance constraint. However, such a Hamiltonian path covering all switches may not always exist in the given switch allocation or require prohibitively high computational complexity to obtain. Therefore, the first objective of the switch routing is to maximize the number of switches covered by the resulting routing path. For the uncovered switches, we add branches to the resulting path (also called *trunk path* in later discussion) to propagate the wake-up-request signal to them. However, the switches on the branches cannot send back a signal back to the wake-up-acknowledge signal. Figure 7 shows an example of adding a branch to the trunk path. As a result, when testing this MTCMOS IC, we cannot ensure whether the MTCMOS switches on the branch can be successfully turned on. This disadvantage on testing is the most important reason why the switch routing avoids the usage of branches and attempts to maximize the number of switches covered the trunk path.

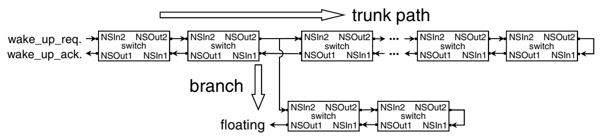

Figure 7: Trunk path and branch for switch routing.

The second objective of the switch routing is to minimize the total length of the trunk path in terms of Manhattan distance. A shorter trunk path can leave more routing resources for the routing of the gated low-V_t cells. Also, a shorter routing path can result in a shorter response time of the wake-up-acknowledge signal.

The problem formulation of the proposed switch-routing framework is summarized as follows.

Input:

- The location of each switch after switch allocation.
- The Manhattan-distance constraint between two connected switches.
- The starting location (the wake-up-req. signal).

Output:

- A trunk path which visits each switch at most once.
- Branches which cover the switches not visited by the trunk path.

Objective:

- First priority: maximize the number of switches covered by the trunk path.
- Second priority: minimize the total length of the trunk path and branches in terms of Manhattan distance.

3.2 MTCMOS Switch Routing Using TSP Solver

Several TSP solvers have been developed in the past to find a Hamiltonian path with minimal length. However, current public TSP solvers (such as Concorde [13], GOBLIN [14], or LKH [15]) are all performed based on a complete graph, where any two nodes are connected to each other and a Hamiltonian path can always be easily found. Due to the Manhattan-distance constraint, the connection graph of MTCMOS switch routing is not complete and hence the existing TSP solver cannot be directly applied to solve the MTCMOS switch routing.

In order to do so, we try the following method based on a modified complete graph. First, we assign each edge's weight as the distance between the two switches of the edge if this distance is smaller than the given Manhattan-distance constraint. Then, to further lead the TSP result to satisfy the Manhattan-distance constraint, we assign an excessively large weight to each edge whose distance between its two switches exceeds the constraint. In our experiment, this excessively large weight is set to the summation of the distance between any two switches. In other words, as long as any of those constraint-violated edges is included in the resulting Hamiltonian path, the length of the Hamiltonian path is guaranteed to be larger than that of a path including no constraint-violated edge. Therefore, while a TSP solver tries to minimize the path length, those constraint-violated edges should be avoided if the TSP algorithm is optimal enough.

Unfortunately, the TSP solvers we tried cannot lead to a Hamiltonian path without going through a constraint-violated edge. We will show the experimental results later in the paper. This inefficiency of using an existing TSP solver to solve MTCMOS routing is actually one of our motivations to develop a new framework for MTCMOS switch routing.

4. PROPOSED FRAMEWORK

Basically, our switch-routing framework applies a greedy-based algorithm to find a minimal Hamiltonian trunk path of switches. The algorithm begins with a given starting point (the wake-up-req. signal) and each time selects the unvisited switch closest to the current switch as the next routed switch. Also, we utilize the following three techniques, the *bridge creation* (Section 4.2), the *switch absorption* (Section 4.3), and the *rectangle routing* (Section 4.4), to enhance algorithm's flexibility of finding a feasible Hamiltonian path and reduce total path length. The data structure representing switch's location is shown in Section 4.1. The overall flow of the proposed switch-routing framework is summarized in Section 4.5.

4.1 Data Structure for Switch's Location

In our switch-routing framework, we use a two-dimensional *position matrix* to record the location of switches. The elements on the same row (column) have the same coordinate in the X (Y) axis. If the value of an element is 1, a switch is located at the element's location. Otherwise, no switch exists on that location. The distance between each pair of adjacent rows (or columns) may not be the same and hence is recorded in another matrix, called *distance matrix*. The search for switches discussed in later subsections is performed based on the position and distance matrices. Note that we do not record the distance between each two switches to speed up the search of the closest switch. This is because the total number of switches in our MTCMOS design may be larger than 100K and the size of a N^2 edge matrix may exceed our system's limitation.

4.2 Bridge Creation

Since a greedy algorithm is applied to select the next routed switch of the trunk path, it is quite often that the resulting path goes to a dead end, from where the distance to the closest unvisited switch exceeds the Manhattan-distance constraint. In such cases, a *bridge* is created to connect the last routed switch S_C to the next routed switch S_N by removing some routed switches and using them as the intermediate switches between S_C and S_N. However, not every routed switch is *removable*. A routed switch S_R is removable if $MD(S_{R-1}, S_{R+1})$ satisfies the Manhattan-distance constraint, where $MD(S_{R-1}, S_{R+1})$ denotes the Manhattan distance between S_R's ancestor switch S_{R-1} and descendant switch S_{R+1}.

In the bridge creation, we choose the next intermediate switch S_R as the removable switch satisfying the following two conditions: (1) $MD(S_R, S_C)$ is under the Manhattan-distance constraint, and (2) $MD(S_R, S_N)$ is as short as possible. Then we remove the intermediate switch S_R from the trunk path, connect S_C to S_R, and connect S_R's ancestor switch S_{R-1} to its descendant switch S_{R+1}. After such a modification, each connection in the trunk path still can satisfy the Manhattan-distance constraint and the last routed switch becomes S_R, which locates much closer to S_N than the original S_C. If the distance between S_R and S_N still exceeds the constraint, we repeat the above actions to find another proper intermediate switch closer to S_N.

Figure 8 shows an example of creating a bridge from S_{24} to S_N, where the sequence of the routed switches is S_1, S_2, ..., S_{24}, and $MD(S_{24}, S_N)$ exceeds the constraint. In Figure 8(a), we select the intermediate switch S_{15} following the above two conditions. Then we remove S_{15}, connect S_{24} to S_{15}, and connect S_{14} to S_{16} in Figure 8(b). After that, the last routed switch becomes S_{15} but $MD(S_{15}, S_N)$ still exceeds the constraint. So, we select anther intermediate switch S_6 in Figure 8(c). Next, we remove S_6, connect S_{15} to S_6, and connect S_5 to S_7 in Figure 8(d). Now, the distance between the last routed switch S_6 and S_N can satisfy the constraint, and hence we can directly connect S_6 to S_N.

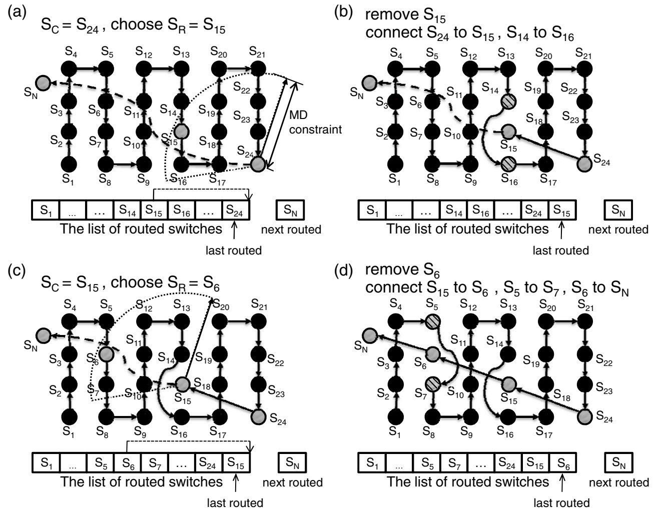
(a)
S_C = S_24 , choose S_R = S_15
MD constraint
The list of routed switches
next routed
last routed
(b) remove S_15
connect S_24 to S_15 , S_14 to S_16
(c)
S_C = S_15 , choose S_R = S_6
(d) remove S_6
connect S_15 to S_6 , S_5 to S_7 , S_6 to S_N

Figure 8: An example of bridge creation.

If the closest unvisited switch S_N to the current switch S_C is on the opposite side of a large hard macro such as Figure 9, then the above process of bridge creation may fail because it tends to select the intermediate switches locating between

S_C and S_N. In this situation, we have to detour the way of selecting the intermediate switches along the boundary of the hard macro, such as Figure 9. Therefore, we select few removable routed switches at the rectangle's corners as temporary target switches, such as S_{T1} and S_{T2} in Figure 9. By creating bridges to temporary target switches, we can successfully connect to the real target switch S_N. Note that, in this detour situation, the estimated distance from S_C to S_N should be the summation of $MD(S_C, S_{T1})$, $MD(S_{T1}, S_{T2})$, and $MD(S_{T2}, S_N)$, instead of $MD(S_C, S_N)$. Thus, before really creating a bridge to S_N, we check whether the distance from S_C to its second closest unvisited switch is shorter than this estimated distance to S_N. If yes, we change the next routed switch to the second closest one. In addition, to complete the above detour, the location and the dimensions of each hard macro need to be recorded in advance so that we can efficiently check their existence before bridge creation.

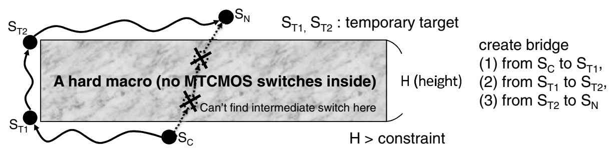

Figure 9: Detour in bridge creation.

4.3 Switch Absorption

At the late phase of the switch routing, some unvisited switches may scatter over the IC, such as S_N in Figure 10(a). If we use bridge creation to connect it, the total path length may increase a lot, such as Figure 10(b). To economically connect a dangling unvisited switch S_N, we can break an existing edge from S_1 to S_2 and then add another two edges from S_1 to S_N and from S_N to S_2, as shown in Figure 10(c). This operation looks like to absorb an unvisited switch into the routed trunk path and hence is called *switch absorption.* The premise to perform this switch absorption is that both $MD(S_1, S_N)$ and $MD(S_N, S_2)$ are under the Manhattan-distance constraint.

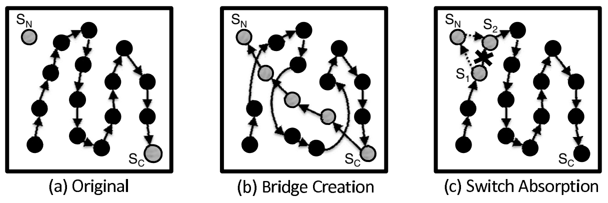

Figure 10: Difference between bridge creation and switch absorption for routing a dangling switch.

To maximize the benefit of using switch absorption, two key problems remain to be solved : (1) when to apply switch absorption instead of bridge creation; (2) which switch to be absorbed first. To solve the first problem, we need to estimate the cost of performing each of switch absorption and bridge creation. The path length increased by creating a bridge to a target switch is usually higher than absorbing one switch. However, after the bridge is created, we may be able to route a group of unvisited switches close to the target switch with short edges, which can compensate the cost of creating the bridge. Therefore, the cost of creating a bridge should be the average edge length added before the next bridge creation (including the length of the current bridge).

In fact, at the early phase of switch routing, a lot unvisited switches can be routed after a bridge is created. Thus, we will not consider the use of switch absorption at all until a high percentage of switches are already routed (more specifically, 98% in our experiment), which can save the computational overhead of comparing the costs of bridge creation and switch absorption. Once using switch absorption, we stop using bridge creation and absorb all the remaining unvisited switches. Therefore, we compare the cost of creating a bridge with the average added length of absorbing each of the remaining switches.

The most possible way to absorb an unvisited switch is to break the edge either coming to or starting from its closest routed switch. Thus, the cost of absorbing the unvisited switch can be approximately estimated by the shorter added length of choosing either of the above two edges for absorption. To collect this absorption cost for all unvisited switches from scratch is expensive. Hence, for each unvisited switch, we record its closest routed switch, its distance to the closest routed switch, and its absorption cost. Each time a new switch is routed, we check whether its distance to each unvisited switch is smaller than its recorded closest distance. If yes, we update the closest-routed-switch information for the unvisited switch. If no, no change is made. Note that we start to record the above closest-routed-switch information when 98% of the switches are already routed. Its computational overhead is limited.

As to the second problem, the ordering of absorbed switches may affect its total added path length since the new absorbed switch may form a better edge to break for the later absorbed switches. Thus, we always absorb the switch with the shortest distance to its closest routed switch, leaving the other switches a chance of shortening their distance to their closest routed switch. Also, we maintain the same closest-routed-switch information as above to determine the next absorbed switch.

4.4 Rectangle Routing

For a MTCMOS design, majority of the switches are placed regularly based on a placement pattern as shown in Section 2.2. To take advantage of this regularity, we first identify several maximal-size rectangles in which the switches are all regularly placed, meaning that the X-axis (or Y-axis) distance between any two adjacent switches is the same. Then we route all the switches inside a rectangle at once. In fact, the "rectangle" defined in our framework does not have a real rectangular shape physically due to the placement patterns, but we conceptually view it as a rectangle as shown in Figure 11. For such rectangles, a minimum Hamiltonian path covering all rectangle's switches can always be found efficiently.

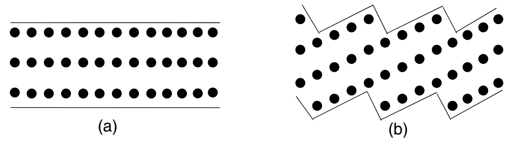

Figure 11: (a) A conceptual rectangle used in our rectangle routing, and (b) the real location of the switches inside the rectangle.

In our framework, we use only the following three types of Hamiltonian-path routing to route the switches within a rectangle. Because the X-axis unit length (X-axis distance between two adjacent switches) is much longer than the Y-axis unit length in the used placement pattern, we try to avoid the traverse in X-axis as much as possible. With these three types of Hamiltonian-path routing, we can guarantee that the resulting Hamiltonian path within the rectangle has the shortest length given the input switch (the first routed switch). Once the routing type and the input switch is determined, its output switch (the last routed switch in the rectangle) is determined as well.

Figure 12 shows the three types of Hamiltonian-path routing and their corresponding path length on a MxN rectangle. When the input switch is on the top (or bottom) side of the rectangle, we apply the Type-1 routing. To use the Type-1 routing, P has to be odd, where P means that the input switch is the Pth switch on the top side counting from the closer right or left side. Therefore, an input switch with an even P will not be considered when the routing starts from the top side.

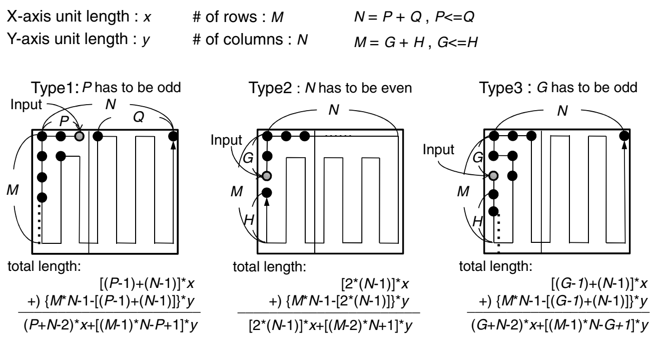

Figure 12: Rules for routing a single rectangle.

When the input switch is on the left (or right) side, we choose the proper routing between type-2 and type-3 routing. To apply type-2 routing, N has to be even, where N is number of rows. To apply type-3 routing, G has to be odd, where G means that the input switch is the Gth switch on the left side counting from the closer top or bottom side. If both type-2 and type-3 routing cannot be applied to an input switch, then that input switch will not be considered when the routing starts from the left side. If both type-2 and type-3 routing can be applied to an input switch, then both types will be considered for that input switch. Note that the total length for each type routing listed in Figure 12 is only an approximation (lower bound actually). The sawtooth-like top and bottom sides requires a little extra length in Y axis, which can be computed by a slightly more complicated equation. We omit this equation due to the page limitation.

In our switch-routing framework, the rectangle routing is performed before the switch absorption. Also, the rectangle routing is performed after (1) most switches outside the rectangles are routed and (2) the creation bridge requires a higher cost than the average cost of switch absorption. The rectangle routing first determines the ordering of rectangles to be routed using a greedy algorithm. This greedy algorithm starts from the last routed switch S_C, and each time selects the next routed rectangle whose center is closest to the center of the current rectangle (or S_C at the first time). We route one rectangle at a time based on this rectangle ordering.

For each rectangle to be routed, we choose the best input switch along with a proper type of Hamiltonian-path routing which can minimize the summation of the following three distances: (1) the distance from the last routed switch S_C to the rectangle's input switch, (2) the total path length within the rectangle according to the adopted routing type, and (3) the distance from the output switch of the current rectangle to the center of the next routed rectangle (as shown in Figure 13).

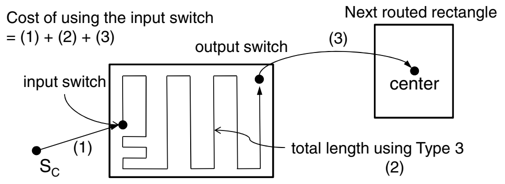

Figure 13: Choose the best input switch for a rectangle.

Note that the rectangles are viewed as hard macros before the rectangle routing. Thus, we need to determine which maximal-size rectangles are preserved for the rectangle routing at the beginning of the switch routing. More rectangles preserved for switch routing imply that more regularly placed switches can be routed in a local minimal way. However, at the same time, more obstacles exist when routing the switches outside the rectangles. So far we have not found an effective method to estimate which group of preserved rectangles can result in a minimal total length. Therefore, we first start from using no rectangle, and then each time add the rectangle covering most switches until adding the rectangle cannot further reduce the total length. Our experimental result will show that the proposed switch-routing framework is efficient enough to afford the iterative trials. In addition, when we estimate the average cost of switch absorption, the switches within the rectangles are considered routed since the switch absorption is performed after the rectangle routing.

4.5 Overall Flow

Figure 14 summarizes the overall flow of the proposed switch-routing framework. In fact, our algorithm cannot guarantee that a feasible Hamiltonian trunk path can be always found for a given switch allocation. The switches that cannot be absorbed are connected by using branches. Fortunately, no branch is required in our experiments so far.

5. EXPERIMENTAL RESULT

5.1 Results of Proposed Framework

In this subsection, we first perform the proposed switch-routing frameworks on four industrial MTCMOS designs based on a 65nm coarse-grain MTCMOS technology [11] with 6 metal layers. Among those designs, Case 1 is already in production, containing 320 hard macros (including both power-gated and always-on macros) and more than 1-million instances (around 600K power-gated instances). The switch allocation is done semi-automatically with the back-end tool, *Blast Fusion* [12]. The Manhattan-distance constraint is set to $150\mu m$.

Table 1 first shows the result of the proposed switch-routing framework using different numbers of rectangles. In

```
Algorithm: SwitchRouting
# define
$MD(s1, s2)$: Manhattan distance between $s1$ and $s2$
$Nearby(s)$: the switch closest to $s$
$MD\_constrain$: Manhattan distance constraint
$unvisit\_ratio$: the ratio of unvisited switches no in rectangles
$threshold$: user-specified ratio determining when to consider
             switch absorption
$cost\_bc(S_c, S_n)$: cost of creating a bridge from $S_c$ to $S_n$
$cost\_sa\_avg()$: cost of absorbing $S_n$
$Plist$: List of path switches
1  begin
2    Build position matrix and matrix for all switches
3    Build hard macro records
4    $S_c$ = starting switch;
5    Append $S_c$ to $Plist$;
6    while (any unvisited switch exists) {
7      $S_n = Nearby(S_c)$
8      if ($MD(S_c, S_n) < MD\_constrain$) {
9        append $S_n$ to $Plist$; $S_c = S_n$;
10     }
11     else if ($unvisit\_ratio > threshold$ ||
12            $cost\_bc(S_c, S_n) < cost\_sa\_avg()$) {
13       create a bridge from $S_c$ to $S_n$;
14       append $S_n$ to $Plist$; $S_c = S_n$;
15     }
16     else if (rectangles not routed yet) {
17       route rectangles from $S_c$;
18       $S_c$ becomes the last routed switch in rectangles;
19     }
20     else
21       break
22   }
23   try to absorb remaining switches into $Plist$ if possible;
24   create branches to connect remaining switches;
25 end
```

Figure 14: The overall algorithm of the proposed switch-routing framework.

Table 1, Column 2 and 3 list the total chip size and total number of MTCMOS switches, respectively. Column 4, 5, and 6 list the number of rectangles in use, the percentage of switches inside the rectangles, and the total Manhattan distance of the trunk path, respectively. A shadowed slot indicates the shortest path length of using different numbers of rectangles. Column 7 lists the percentage of switches covered by the trunk path. Column 8 lists the total runtime in seconds.

As the result shows, the proposed switch-routing framework can always find a feasible trunk path covering all the switches. Also, using only the largest one or two rectangles can already result in the shortest trunk path. It is because majority of the switches are regularly placed in real designs so that the largest two rectangles can cover around 50% of the total switches. In addition, the reported runtime demonstrates that the complexity of the proposed framework is scalable to large industrial designs, where the longest runtime is around 1.15 hours for routing 173K MTCMOS switches. Even we try the number of used rectangles from 0 to 3 respectively, the total runtime for the largest case is around 2.4 hours.

5.2 Compared with Vendor Solution

In this subsection, we compare the proposed framework with a rough solution provided by an EDA vendor, which is a script file operating on the design database used in a back-end tool, not a tool's standard built-in function. Also, this solution does not consider the Manhattan-distance constraint. However, it is the only switch-routing solution we can get from our EDA vendor. Actually, one of our motivations to start developing the proposed switch-routing framework is from seeing the inefficiency of this rough vendor solution. The comparison results are reported in Table 2.

design	chip size (mm^2)	# of switches	# of rect.	switch % in rect.	total MD (μm)	trunk path coverage	runtime (sec)
Case 1	13.595	17523	0	0	257385	100%	107
			1	40.10	256425	100%	90
			2	53.03	256349	100%	71
			3	55.41	257042	100%	73
Case 2	13.552	20593	0	0	325196	100%	102
			1	39.14	323945	100%	102
			2	50.17	327959	100%	112
			3	51.27	326792	100%	80
Case 3	29.426	57035	0	0	853093	100%	373
			1	40.16	852055	100%	181
			2	58.09	850614	100%	111
			3	72.31	854374	100%	150
Case 4	66.234	173420	0	0	2570350	100%	4139
			1	39.04	2545080	100%	1512
			2	46.06	2548310	100%	1589
			3	52.14	2550770	100%	1374

Table 1: Results of the proposed switch-routing framework on 4 industrial MTCMOS designs.

In Table 2, Column 3 and 4 list the total Manhattan distance of the trunk path and its number of violations against the Manhattan-distance constraint, respectively. Column 5 and 6 list the total wire length and the number of vias in use after detail route, respectively. Column 7 lists the path's response time from the wake-up-request signal to the wake-up-acknowledge signal. As the result shows, the proposed switch-routing framework performs better than the vendor's solution at every reported item. Note that the switch delay with an oversized output loading is estimated by using the linear extrapolation. The actual response time for the vendor's solution might be larger than the reported number shows.

design	method	total MD (μm)	# of MD violation	wire length (μm)	# of via	response time (μ s.)
Case 1	vendor	922,634	382	1,907,690	142,625	7.87
	proposed	256,349	0	573,513	142,119	6.62
Case 2	vendor	908,123	317	1,892,643	168,948	8.53
	proposed	323,945	0	720,615	167,933	7.67
Case 3	vendor	1,991,841	429	4,193,585	475,952	21.85
	proposed	850,614	0	1,904,906	464,989	19.79
Case 4	vendor	5,356,878	580	11,346,593	1,438,786	63.28
	proposed	2,545,080	0	5,715,291	1,416,614	58.50

Table 2: Comparison between the propose framework and a vendor solution.

5.3 Compared with TSP Solver

In this subsection, we attempt to solve the switch-routing problem by applying a TSP solver based on a modified complete graph as described in Section 3.2. Since we set an excessively large weight to each edge whose distance between its two switches exceeds the constraint, the TSP solver should avoid passing through such a constraint-violated edge while minimizing the total path length. If the TSP solver is optimal enough, no constraint-violated edge will be visited and hence a feasible Hamiltonian path can be found.

We first implement a simple greedy TSP algorithm, denoted as $TSP1$, which selects the unvisited switch closest to the current switch as its next ordered switch each time. Also, $TSP1$ will try several different initial switches individually. Among those trails, $TSP1$ reports the Hamiltonian path covering the least number of constraint-violated edges. The number of trials used in $TSP1$ is 0.1% of the total switches. Note that the search of the closest unvisited switch is done based on the same data structure as the proposed method (Section 4.1), such that we can avoid the use of a huge N^2 matrix for storing all edge's weight, where N is the total number of switches. Thus, the search of the closest unvisited switch in $TSP1$ may be relatively slow compared to the use of a N^2 edge matrix.

Table 3 compares the total Manhattan distance of the resulting Hamiltonian path, the number of constraint-violated edges, and the runtime between $TSP1$ and the proposed framework. As the result shows, the listed total Manhattan distance between $TSP1$ and the proposed framework is close. However, the path reported by $TSP1$ contains at least 24 more constraint-violated edges for each benchmark design. To eliminate all the constraint-violated edges from the reported path, a significant amount of Manhattan distance is needs be added to form a feasible Hamiltonian path. Also, the runtime of $TSP1$ is much longer than that of the proposed framework, especially for the largest design.

design	method	total MD (μm)	# of MD violation	runtime (sec)
Case 1	TSP1	259,218	24	1008
	proposed	256,349	0	71
Case 2	TSP1	321,048	28	1087
	proposed	323,945	0	102
Case 3	TSP1	857,110	26	10298
	proposed	850,614	0	111
Case 4	TSP1	2,550,120	32	193379
	proposed	2,545,080	0	1512

Table 3: Comparison between the propose framework and a greedy-based TSP solver.

Next, we try to apply a state-of-the-art TSP solver [13] to solve the switch-routing problem and check whether this advanced TSP solver can generate a feasible Hamiltonian path without going through any constraint-violated edge. The TSP solver [13] can be obtained from public domain and has been applied to solve several optimization problems [16] [17] [18]. Also, this TSP solver [13] can always iteratively fine-tune an existing solution to obtain a better solution. However, this TSP solver [13] requires a two-dimensional matrix to store all edges' weight of the complete graph.

Table 4 first lists the size of the edge matrix. For Case 3 and 4, the size of their edge matrices exceeds the limitation of the TSP solver [13] (as well as the main-memory size of our system) and hence no result can be obtained. For Case 1 and 2, the resulting path reported by the TSP solver [13] still contains few edges violating the constraint after running for more than 24 hours. This result demonstrates that a feasible short Hamiltonian path, which can be efficiently and effectively obtained by the proposed switch-routing framework, is not easy to be obtained by using a general TSP solver. It also shows the advantage of developing a heuristic algorithm which is specialized for solving MTCMOS switch routing and can utilize the physical-layout information of the MTCMOS switches.

6. CONCLUSION

In this paper, we proposed a switch-routing framework, which utilizes the techniques of the bridge creation, the switch absorption, and the rectangle routing to simultaneously minimize the trunk-path length and satisfy the Manhattan-distance constraint. The experimental result demonstrates its efficiency and effectiveness by comparing to a vendor solution and TSP solvers based on four industrial MTCMOS designs. This framework is currently applied in an industrial design house.

design	Case 1	Case 2	Case 3	Case 4
required edge-matrix size	1.3GB	1.9GB	14.3GB	141.0GB
# of left MD violations	2	2	N.A.	N.A.

Table 4: Required matrix size and the number of left violations against the Manhattan-distance constraint after running 24 hours for [13].

7. REFERENCES

[1] Z. Liu and V. Kursun, Charge Recycling Between Virtual Power and Ground Lines for Low Energy MTCMOS, *IEEE/ACM International Symposium on Quality Electronic Design*, pp.239-244, March 2007.

[2] E. Pakbaznia, F. Fallah, and M. Pedram, Charge Recycling in MTCMOS Circuits: Concept and Analysis, *Design Automation Conference*, pp.97-102, July 2006.

[3] S. Kim, S. V. Kosonocky, D. R. Knebel, and K. Stawiasz, Experimental Measurement of a Novel Power Gating Structure with Intermediate Power Saving Mode, *International Symposium on Low Power Electronics and Design*, pp.20-25, Aug. 2004.

[4] C. Long and L. He, Distributed Sleep Transistors Network for Power Reduction, *Design Automation Conference*, pp.181-186, July 2003.

[5] J. Kao, S. Narendra, and A. Chandrakasan, MTCMOS Hierarchical Sizing based on Mutual Exclusive Discharge Patterns, *Design Automation Conference*, pp.495-500, June 1997.

[6] C. Hwang, C. Kang, and M. Pedram, Gate Sizing and Replication to Minimize the Effects of Virtual Ground Parasitic Resistances in MTCMOS Designs, *IEEE/ACM International Symposium on Quality Electronic Design*, March 2006.

[7] M. Anis, S. Areibi, and M Elmasry, Design and Optimization of Multithreshold CMOS (MTCMOS) Circuits, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp.1324-1342, Volume 22, Issue 10, Oct. 2003.

[8] A. Ramalingam, A. Devgan, and D. Z. Pan, Wakeup Scheduling in MTCMOS Circuits Using Successive Relaxation to Minimize Ground Bounce, *Journal of Low Power Electronics*, pp.1-8, Vol.3, No.1, 2007.

[9] H. Jiang and M. Marek-Sadowska, Power Gating Scheduling for Power/Ground Noise Reduction, *Design Automation Conference*, pp.980-985, June 2008.

[10] A. Abdollahi, F. Fallah, and M. Pedram, A Robust Power Gating Structure and Power Mode Transition Strategy for MTCMOS Design, *IEEE Trans. on Very Large Scale Integration Systems*, pp.80-89, Volume 15, Issue 1, Jan. 2007.

[11] Taiwan Semiconductor Manufacturing Company, Ltd., TSMC Reference Flow 7.0, 2007.

[12] Magma Design Automation, Blast Fusion User Guide Version:2005.03, June 2007.

[13] D. Applegate, R. Bixby, V. Chvatal, and W. Cook Concorde TSP Solver, http://www.tsp.gatech.edu/concorde/index.html.

[14] C. Fremuth-Paeger, B. Schmidt, and B. Eisermann, GOBLIN: A Graph Object Library for Network Programming Problems, http://www.math.uni-augsburg.de/ fremuth/goblin.html.

[15] K. Helsgaun, LKH: An Effective Implementation of the Lin-Kernighan Heuristic for solving TSP, http://akira.ruc.dk/ keld/research/LKH/.

[16] D. Aldous and A. G. Percus, Scaling and Universality in Continuous Length Combinatorial Optimization, *PROC. Nat. Acad. Sci. USA 100 (20)*, pp.11211-11215, 2003.

[17] D. Applegate, W. Cook, S. Dash, and A. Rohe, Solution of a Min-Max Vehicle Routing Problem, *INFORMS Journal on Computing 14 (2)*, pp.132-143, 2002.

[18] G. Gutin, H. Jakubowicz, S. Ronen, and A. Zverovitch, Seismic Vessel Problem, *Communications in DQM 8*, pp.13-20, 2005.