

基于 AIG 的多级逻辑电路延迟近似优化

赵维凯, 于宗源, 王伦耀*

(宁波大学 信息科学与工程学院, 浙江 宁波 315211)

摘要: 在对多级逻辑电路延迟进行优化中, 提出了一种针对关键路径中节点输出的近似替换方法, 用于实现延迟优化. 提出的算法先建立待优化电路的关键路径集合, 然后通过选取每一条关键路径中错误率影响最小的节点构成待优化节点集, 再结合提出的节点输出近似替换技术, 在错误率约束下实现节点删除和关键路径压缩, 进而达到多级逻辑电路延迟优化. 提出的算法用 C++ 和 ABC 工具内置命令编程实现, 使用 ISCAS85 以及 LGSynth91 电路进行测试. 实验结果显示, 与已提出的常量替换方法相比, 面积和延迟优化效果分别提升 22.96% 和 31.49%. 同时相较于最新提出的针对延迟优化的算法, 在延迟优化效果相近的情况下, 算法运行时间上有 61.88% 的提升.

关键词: 近似计算; 逻辑优化; 误差约束; 多级电路; 延迟优化

中图分类号: TN431.2

文献标志码: A

文章编号: 1001-5132 (2023) 01-0035-07

随着晶体管技术达到纳米级, 集成电路设计越来越复杂, 通过传统设计方法提高电路的性能变得愈发困难^[1]. 另一方面, 许多新兴应用程序, 例如机器学习、数据挖掘/分析和图像处理等, 其本质上是容错的. 在这种情况下, 近似计算作为一种新的电路设计范式应运而生^[2]. 其基本思想是在保证电路正常工作的情况下, 通过引入误差来修改其电路结构, 从而获得更小的面积、更低的功耗及延迟. 近似计算主要分为近似电路设计以及近似逻辑综合(Approximate Logic Synthesis, ALS)两个方面, 前者目前主要通过人工设计, 对精确加法器、乘法器等电路进行近似优化^[3], 而后者旨在给定误差约束条件下, 为容错应用自动生成最佳近似电路^[4].

近年来, ALS 得到了研究者的广泛关注, 现有研究工作表明^[5-10], 利用近似计算技术, 在引入较小计算误差的情况下, 能显著地提高电路性能. 文献[5]通过寻找两个功能近似的信号, 用一个替换另一个来优化电路. 文献[6]针对文献[5]提出了批

量的精度可配置误差估计方法, 在相同误差约束下获得了更好的优化结果, 同时缩短了程序运行时间. 文献[7]提出了一种基于深度学习的综合流, 通过在映射过程中移除或者替换网络上的门实现电路功耗优化. 文献[8]通过搜索、获取电路中的 ACS (Approximate Care Set), 并对 ACS 中的节点进行近似替换, 从而实现面积优化. 文献[9]通过获取电路关键路径的切割集, 对其采用常量替换的方式实现电路性能优化, 但常量替换的方式可能会使电路误差激增从而无法在给定约束下获得最优结果. 文献[10]针对关键路径中的节点构建临界误差网络, 通过最小割的方式寻找到最优节点进行常量替换, 从而实现电路延迟优化, 但由于部分电路对应的解空间较大, 其构建网络过程会耗费大量运算时间.

然而, 目前所提出的 ALS 方法主要集中在电路面积上的优化. 虽然这些 ALS 方法在优化面积的同时也会降低电路延迟, 但其在改善电路延迟方面的潜力并没有被完全发掘. 对于实时信号处

收稿日期: 2022-03-22.

宁波大学学报(理工版)网址: <http://journal.lg.nbu.edu.cn/>

基金项目: 国家自然科学基金(62131010, 61471211, 61871242).

第一作者: 赵维凯(1997-), 男, 浙江温州人, 在读硕士研究生, 主要研究方向: 集成电路综合和优化. E-mail: 745112451@qq.com

*通信作者: 王伦耀(1972-), 男, 浙江宁波人, 教授, 主要研究方向: 集成电路综合和优化. E-mail: wanglunyao@nbu.edu.cn

理等应用,它们是容错的,但对延时的要求是严格的^[11].对于这类应用,延迟优化将成为主要目标.由于对电路全局进行近似优化十分困难,以往的ALS方法通常对电路中某一个或几个门不断进行局部近似变化(Local Approximate Change, LAC),直至给定的误差约束.在这些方法中,由于节点数量等效于电路面积且每一个节点对于电路面积的贡献近似相同,可以认为在面积优化时任何节点的删除都会带来电路面积的减小.但这与电路的延迟优化思路是不相同的,因为延迟是由电路的关键路径决定的,关键路径指的是信号由输入端传递到输出端所经过的最长路径,且电路的关键路径往往不止一条,单纯对某一条关键路径进行LAC往往无法使得电路延迟降低.因此,在每一次优化过程中要关注电路中所有的关键路径,即同时缩短每一条关键路径长度.为此,本文提出一种针对电路延迟优化的ALS方法.首先,根据所给定电路获取关键路径节点集.其次,针对关键路径节点采取不同于常量替换的LAC方式.本文以错误率作为误差衡量标准,其估算方式采用蒙特卡洛算法^[12],最后在错误率约束下不断迭代获得最优电路.实验结果表明,与最近所提出的ALS方式相比,本文在延迟优化方面效果显著,同时程序运行时间也存有优势.

1 与非图及误差评价指标

1.1 与非图

由于“与”“非”运算可以构成逻辑运算完备集,因此任何一个逻辑表达式都可以转换为“与”“非”运算的表达式.与非图(AND-Inverter-Graph, AIG)就是一种仅包含“与”和“非”运算,用来表示逻辑功能的有向无环图^[13].AIG由二输入与门、反相器、有向边和输入输出端一同构成.以图1所示的C17电路对应的AIG为例,图中正三角表示输入(Primary Input, PI),倒三角表示输出(Primary Output, PO),圆圈表示与门. AIG中的有向连线表示节点之间存在信号输出输入关系,其中虚线表示对该节点的输出信号取反,相当于接入反相器.另外,为了方便描述,通常给AIG中的节点、PI和PO进行编号.图1中编号1~7为输入,8~13为节点,22和23为输出.

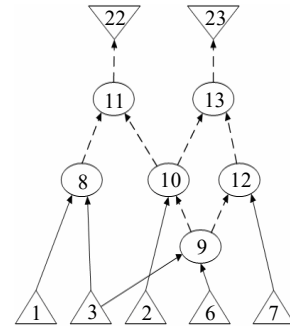


图1 C17电路AIG

一般情况下,AIG中PI到PO之间存在多条路径,其中包含节点最多的路径称为关键路径,关键路径中节点数量越多,则表示电路信号传递时间越长,即延迟越大.由于AIG中的每个节点对应一个“与”运算,因此AIG中包含的节点数越多,意味着对应的电路面积也越大.这也是基于AIG的逻辑优化中常利用AIG节点数和关键路径长度作为电路面积和延迟评估标准的原因^[14].

1.2 误差评价指标

采用近似计算技术的逻辑优化都必须满足给定的近似度约束.近似度有多种衡量方式,本文将错误率作为电路的误差指标.

近似逻辑函数的错误率定义为造成近似函数与原函数输出不同的输入组合占有所有输入组合的比例.对于一个 n 输入, m 输出的逻辑函数 $f(\bar{X})$,令 $\bar{X} = \{x_0, x_1, \dots, x_{n-1}\}$ 为输入变量集合, $f_{ap}(\bar{X})$ 是 $f(\bar{X})$ 的近似函数, R_E 为错误率,则

$$R_E = \frac{M[f(\bar{X}) \neq f_{ap}(\bar{X})]}{2^n} \times 100\%, \quad (1)$$

式中, $M[f(\bar{X}) \neq f_{ap}(\bar{X})]$ 表示在所有输入组合中,使得 $f(\bar{X}) \neq f_{ap}(\bar{X})$ 的输入组合数.

2 多级逻辑电路延迟优化算法

本文的多级逻辑电路延迟优化通过AIG实现,并用AIG的关键路径长度来衡量逻辑电路的延迟,通过减少AIG关键路径节点数实现电路延迟优化.

2.1 获取关键路径不重复节点集

不同于面积优化,在延时优化中,首先必须考虑关键路径上节点的删除,才能达到优化延时的目的.为此,在延时优化过程中,必须先获取当前电路的所有关键路径节点集.

对于电路延迟上的优化,单单缩短一条关键

路径往往是不够的. 如图 1 所示的 C17 电路, 它一共存在节点编号为 {9,10,11}, {9,12,13}, {9,10,13} 的 3 条关键路径集合. 例如单独优化节点 11, 则无法实现电路延迟的优化. 而若同时优化节点 11、13, 则可以使电路延迟由 3 缩短到 2, 但意味着更多节点被优化而导致错误率增加.

根据 C17 的关键路径可以发现, 所获得的关键路径存在许多重复的节点. 由于本文后续需要对所有关键路径节点单独进行 LAC 操作并获得对应的错误率, 这会使得处在不同路径上的同一节点被重复计算. 例如节点 9 存在于 C17 电路所有的关键路径中, 在计算错误率时会进行 3 次重复的计算, 而每一次优化所产生的错误率结果是相同的, 从而造成程序运行时间增加. 由此本文提出了一种获取不重复的关键路径节点方法, 伪代码如下, 其中 G 为输入的 AIG 电路.

算法 1 Collect_no-repeat_pathcone(G)

Input: an AIG G ;

Output: no-repeat critical path node sets;

$POs = \text{Col_AIG_POs}(f.AIG)$; //step1

foreach PO then

if $PO \rightarrow level == \text{GetNtklevel}(G)$ then// step2

while $node \rightarrow level > 1$ do

if $node \notin nodecone$ then

nodecone = SaveCriticalPath($node$); //step3

$node = \text{GetNextNode}(PO)$; //step4

在 step1 中, 通过 Col_AIG_POs 函数获取当前输入 AIG 电路中的所有输出端节点 POs. 由于电路的 PO 并不一定都是关键路径的起始节点, 因此在 step2 中通过 GetNtklevel 函数获取当前输出端的层级, 判断输出端层级是否与电路层级相同来选取正确的关键路径起始探索节点, 自顶向下搜索所有关键路径集合. 同时, 为了避免关键路径的重复记录, step3 先判断当前节点是否已经存在于关键路径节点集中, 若不存在, 则将该节点进行保存; 若存在, 则不重复记录. step4 中 GetNextNode 函数的作用是通过当前节点获取到关键路径的下一个节点. 由于 AIG 中每一个节点仅有两个对应的扇入, 本算法通过判断两个扇入节点所处层级的大小来决定以哪一个节点作为关键路径节点. 若两者不一样大, 则将层级大的节点作为关键路径节

点继续进入循环, 层级小的节点视为非关键路径节点而舍弃; 若两者一样大, 则将两个扇入节点都作为下一次关键路径的节点进行搜索.

以 C17 电路为例说明算法 1, 从 PO 端开始探索, 首先节点 11、13 的层级与电路层级相同, 将这两个节点作为关键路径探索的首节点:

(1)以节点 11 为首节点进行探索, 节点 8、10 都为节点 11 的扇入, 通过判断节点 10 的 level 为 2, 而节点 8 的 level 为 1, 将节点 10 作为关键路径进行保存. 随后对于节点 10, 同样进行上述判断, 将节点 9 保留为关键路径.

(2)以节点 13 为首节点进行搜索, 节点 10、12 为节点 13 的扇入节点, 而节点 10、12 所处层级都为 2, 则同时对两个节点进行探索. 由于在以节点 11 为首节点探索时, 节点 10 已经记录于关键路径中, 则不进行重复记录, 且节点 10 的后续探索也已经在上一次记录中全部完成, 可以认为该路径下的所有节点都已经被记录. 对于节点 12 的扇入节点 7、9, 因节点 9 所在层级较高, 并且已经存在于关键路径中, 不进行重复记录.

由此 C17 电路的不重复节点集搜索已完成, 该集合为 {9,10,12,11,13}.

2.2 近似局部变换及错误率估算方法

根据 AIG 节点的特性, 允许某个节点拥有多个扇出端, 进而可以将许多功能相同的节点合并以实现电路性能上的优化. 不同于文献[9-10]中对于节点集或者节点进行常量替换造成目标节点的所有扇入信息丢失, 本文通过将目标节点在关键路径上的扇入节点与扇出节点直接相连, 保留了关键路径上的信息, 仅牺牲了非关键路径上扇入节点输入的信息, 同时又避免了出现常量信号传递到下一个节点造成电路大量节点删除的情况, 从而以更小的错误率实现层级的压缩. 图 2(a)所示为 C17 电路节点 10 使用常量 0 替换结果, 节点 {10,11,13} 删除, 电路延迟由 3 缩短为 2. 图 2(b)为仅对节点 10 所在的关键路径进行 LAC 的结果. 由于节点 10 的扇入节点 2 和 9 中仅有节点 9 存在于关键路径中, 且节点 11、13 同为节点 10 的关键路径上的扇出, 据此将节点 11、13 分别与节点 9 相连, 忽略节点 2 的输入信息, 仅删除了节点 {10,12}. 相比于常量替换方法, 本文的方法对电路结构的

影响较小, 从而控制错误率的增加.

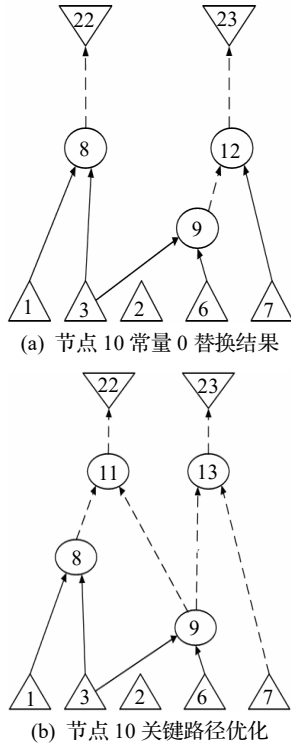


图 2 关键节点 10 不同优化方式

在本文算法中, 针对每一个关键路径节点, 都采用上述 LAC 方法获取对应的错误率. 在错误率计算方面, 本文将原始电路保存, 并将其与近似电路构建 Miter 电路^[15]后, 采用文献[8]所提供的基于蒙特卡洛算法估算电路所对应的错误率.

2.3 基于 AIG 的逻辑电路延迟近似优化算法

本优化算法是基于错误率约束下的迭代优化算法, 主要优化目标为在错误率约束下减少关键路径上的节点数量. 采取的策略是每一次迭代时同时缩短每一条关键路径长度, 以实现电路延迟优化. 对于每一条关键路径, 删除任意一个节点对延迟的贡献都是相同的, 使得搜索最合适的 LAC 节点集的空间非常庞大. 假设电路的关键路径长度为 n , 电路存在 m 条关键路径, 则需要模拟 n^m 次才可以得到最合适的 LAC 节点集. 当电路规模较大时, 通过上述方式来获得最优候选集将会产生巨额的计算量. 为提高运算速度, 本文通过选取每一条关键路径在单独 LAC 过程中产生错误率最低的节点, 近似地将该节点作为该条关键路径中的最优解节点. 该方法虽然无法获取到最优的近似结果, 但是极大程度上缩短了寻找合适节点集的

过程, 减少了程序运行时间.

算法 2 为错误率约束下基于近似计算技术的 AIG 电路延迟优化代码, 其中 G 为输入的 AIG 原始电路, et 为优化过程中电路允许的最大错误率.

算法 2 Delay_opt(G, et)

Input: an AIG G , an error threshold et ;

Output: Approximate circuit;

G_{org} = AIG_Duplication(G); //step1

While ($G_{er} < et$) do

 nodecone = Collect_no-repeat_pathcone(G);

 foreach $node$ in nodecone then

er = GetLAC_ER($G, G_{org}, node$); //step2

 LACcone = GetLACnodecone($G, er, node$); //step3

G_{er} = AIG_LAC($G, G_{org}, LACcone$); //step4

在 step1 中首先将最原始电路保存, 用于每一次进行 LAC 操作后计算对应的错误率. 在 step2 中, 先获取待优化电路的不重复关键路径节点集合, 并通过 GetLAC_ER 函数计算每一个不重复节点单独进行 LAC 后对应的错误率. 在 step3 中选出每一条关键路径对应的错误率最低节点, 构成 LAC 节点集. 利用 step3 得到的 LAC 节点集, 在 step4 中对该节点集每一个节点同时进行 LAC 操作, 计算当前电路相对原始电路的错误率. 若错误率超出约束, 则放弃本次尝试, 将上一次优化的电路作为最终结果, 输出错误率以及近似优化电路; 若没有超出错误率, 则继续进行新一轮循环.

同样以 C17 电路为例进行说明, 由于该电路规模较小, 本文仅以一轮模拟进行演示. 根据算法 1 得到 C17 不重复关键路径集合为 {9, 10, 12, 11, 13}, 计算出单独删除上述节点时对应的错误率分别为 {0.1875, 0.59375, 0.4375, 0.8125, 0.8125}, 选取每一条路径中错误率最小的 LAC 节点, 可以发现节点 9 存在于所有关键路径中, 且该节点在对应的关键路径中错误率都是最小的, 由此仅需对节点 9 进行 LAC. 从图 1 可以看到与节点 9 相连的扇出节点 10、12 都是关键路径上的节点, 而节点 9 的扇入节点都为输入端节点, 由于本文的输入模式为均匀输入, 则随机选择一个输入端与扇出端相连, 同时对两条关键路径进行 LAC, 优化后的结果如图 3 所示. 可以发现, 按照本文的方法, 仅以删除一个节点的代价将电路关键路径长度由 3 降低到了 2.

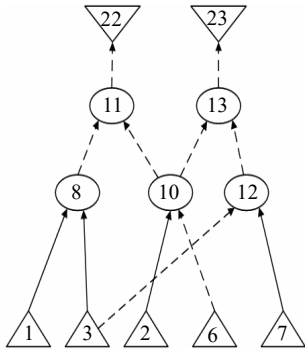


图3 C17电路针对节点9优化结果

3 实验结果分析

本文提出的优化算法用 C++ 语言编程实现并结合逻辑综合工具 ABC^[16] 完成电路延迟优化。运行环境: 处理器频率 2.6 GHz, 内存 16 GB, 操作系统为 Ubuntu 20.04。由于输入电路可能存在冗余结构, 因此对于每一个输入电路都先用 ABC 中的 “resyn2” 优化命令进行预处理, 该命令为 ABC 中内置的电路精确优化脚本命令, 以提高程序优化的效率。

表 1 为本次实验中所需用到的 LGsynth91^[17] 和 ISCAS85 测试电路, 其中的 “面积” 与 “延迟” 为经 ABC 脚本命令优化后, 用 MCNC^[17] 通用标准单元库映射所得到的结果, I/Os 表示该电路对应的输入与输出的端口数量。

表1 本文使用的测试电路信息

电路名称	I/Os	面积	延迟
cm163a	16/5	61	12.3
z4ml	7/4	61	10.1
alu2	10/6	636	51.5
frg1	28/3	179	22.6
unreg	36/16	163	8.0
x2	10/7	74	10.4
count	35/16	203	25.8
term1	34/10	276	23.5
C880	60/26	573	35.0
C1355	41/32	694	26.9
C1908	33/25	668	40.0
C3540	50/22	1 619	48.2
C5315	178/123	2 264	39.1
C7552	207/108	2 640	42.7
alu4	14/8	1 158	49.8

在错误率计算时, 假设所有 PI 输入分布为均匀分布, 通过生成百万级别的输入向量对近似电路进行测试, 使得当前电路计算所得到的错误率与真实错误率相近, 以保证实验结果的准确性。实验中优化程度用优化率 O 表示,

$$O = \frac{C_{\text{org}} - C_{\text{app}}}{C_{\text{org}}}, \quad (2)$$

式中, C_{org} 和 C_{app} 分别为原始电路和近似电路对应的面积或延迟。

表 2 是本文算法与文献[9]的对比结果, 错误率设置与文献[9]相同, 都为 25%, 采用的测试电路为 LGSynth91, 在错误率约束下对 AIG 电路优化完成后, 再通过 “amap” 命令进行映射获取最终面积及延迟。从表 2 中可以发现, 本文方法相较文献[9]在面积和延迟优化上分别提升了 22.96% 和 31.49%。同时文献[9]中有部分电路在 25% 错误率下无法进行任何延迟和面积上的优化, 这是由于其对于关键路径节点或者节点集进行常量替换过程可能会造成大量节点删除, 从而导致电路错误率激增, 无法在错误率约束下实现优化。而本文的 LAC 方式对于电路结构影响较小, 可以很好地控制错误率的增加, 从而获得优于文献[9]的近似电路。

表2 本文算法与文献[9]结果比较 %

电路名称	面积优化		延迟优化	
	本文	文献[9]	本文	文献[9]
cm163a	55.74	53.85	65.04	47.37
z4ml	6.10	0	11.76	0
alu2	22.64	9.73	43.69	0
frg1	93.30	2.49	84.96	0
alu4	24.70	6.01	28.31	0
unreg	39.26	0	31.25	0
x2	41.89	10.81	39.42	1.72
count	43.84	57.85	72.87	79.45
term1	66.67	46.73	61.70	27.03
平均值	43.79	20.83	48.78	17.29

表 3 为本文算法与文献[10]对比的结果, 测试电路为 ISCAS85 电路。文献[10]结果通过该文章在 GitHub 中开源的代码测试, 错误率约束设置为 15%。由于本文与文献[10]都是采用蒙特卡洛算法来估算错误率, 优化电路结果存在随机性, 因此对测试电路都进行了 3 次测试, 取对应平均值作为最

终优化结果.

从表3中可以看出,本文算法面积优化相较于文献[10]减少了1.15%,而延迟优化提升了1.67%,可以看作在面积与延迟优化效果相近,但是本文程序的运行时间相较于文献[10]平均缩短了61.88%.这是由于本文仅将所有关键路径中错误率最小的节点作为本轮优化过程中最优的LAC节点集,并且获取所有不重复关键路径节点并计算其对应的错误率,减少了错误率的重复计算.而文献[10]通过构建临界误差网络来选择该轮最优LAC节点时,相较于本文直接选取错误率最低的节点,将会耗费更多的时间.从实验结果来看,文献[10]与本文算法所得到的电路优化结果近似相同,但其在部分电路测试中无法很好地逼近错误率上界.例如C7552电路,文献[10]仅能得到平均错误率为7.4%的优化结果,无法逼近错误率阈值进而无法获得较好的近似电路.而对于C1355电路优化,两个方式得到的近似优化结果都几乎将所有节点删除,但由于本文算法每一次迭代中节点删除数量相较于常量替换较少,导致全部节点删除需要消耗多轮LAC,因此对于该电路程序运行时间要高于文献[10].

表3 本文算法与文献[10]结果比较

电路 名称	面积优化/%		延迟优化/%		耗时/s	
	本文	文献[10]	本文	文献[10]	本文	文献[10]
C880	1.22	3.26	34.86	11.05	9.49	10.98
C1355	95.39	95.39	96.28	96.28	73.94	63.53
C1908	23.25	20.86	33.25	42.08	24.40	130.59
C3540	4.30	4.86	7.19	10.10	45.01	119.76
C5315	0.80	1.10	7.67	14.66	32.86	123.63
C7552	0.83	15.28	18.74	16.31	42.47	134.53
alu4	13.93	6.99	21.75	17.54	43.03	127.89
平均值	19.96	21.11	31.39	29.72	38.74	101.63

4 结语

本文提出了一种基于AIG在错误率约束下实现多级逻辑电路延迟优化的综合流.该优化方法获取目标电路所有关键路径并收集其中对应的不重复节点.在电路延迟优化过程中,为了避免常量替换造成的电路错误率激增,提出了一种针对关键路径节点的LAC方式.通过该方式,对所有不

重复的关键路径节点集进行LAC并记录其对应的错误率.为了缩短程序运行时间,将每一条关键路径中错误率最低的节点近似地作为最优LAC节点,基于此以迭代的方式不断循环直至超过错误率约束,得到最终的近似电路.本文算法相对已提出的针对面积以及延迟的常量替换方式,在电路性能和程序运行时间上有着显著的提升.

参考文献:

- [1] Xu Q, Mytkowicz T, Kim N S. Approximate Computing: A Survey[J]. IEEE Design & Test, 2016, 33(1):8-22.
- [2] Liu W, Lombardi F, Shulte M. A retrospective and prospective view of approximate computing[J]. Proceedings of the IEEE, 2020, 108(3):394-399.
- [3] Jiang H, Liu C, Liu L, et al. A review, classification, and comparative evaluation of approximate arithmetic circuits [J]. ACM Journal on Emerging Technologies in Computing Systems, 2017, 13(4):1-34.
- [4] Scarabottolo I, Ansaloni G, Constantinides G A, et al. Approximate logic synthesis: A survey[J]. Proceedings of the IEEE, 2020, 108(12):2195-2213.
- [5] Venkataramani S, Roy K, Raghunathan A. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits[C]//Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2013:1367-1372.
- [6] Su S, Meng C, Yang F, et al. VECBEE: A versatile efficiency-accuracy configurable batch error estimation method for greedy approximate logic synthesis[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems [2022-01-09]. <http://doi.org/10.1109/TCAD.2022.3149717>.
- [7] Pasandi G, Peterson M, Herrera M, et al. Deep-PowerX: A deep learning-based framework for low-power approximate logic synthesis[C]//Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, 2020:73-78.
- [8] Meng C, Qian W, Mishchenko A. ALSRAC: Approximate logic synthesis by resubstitution with approximate care set[C]//2020 57th ACM/IEEE Design Automation Conference, San Francisco, 2020:1-6.
- [9] Chandrasekharan A, Soeken M, Große D, et al. Approximation-aware rewriting of AIGs for error tolerant applications[C]//2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2016:1-8.
- [10] Zhou Z, Yao Y, Huang S, et al. DALs: Delay-driven

- approximate logic synthesis[C]//2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018:1-7.
- [11] Wang Y, Wenger S, Wen J, et al. Review of error resilient coding techniques for real-time video communications[J]. IEEE Signal Processing Magazine, 2000, 17(4):61-82.
- [12] Scarabottolo I, Ansaloni G, Constantinides G A, et al. A formal framework for maximum error estimation in approximate logic synthesis[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022, 41(4):840-853.
- [13] Mishchenko A, Chatterjee S, Brayton R. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis[C]//2006 43rd ACM/IEEE Design Automation Conference, San Francisco, 2006:532-535.
- [14] Brayton R, Mishchenko A. ABC: An Academic Industrial-strength Verification Tool[M]//Computer Aided Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010:24-40.
- [15] Brand D. Verification of large synthesized designs[C]//Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), Santa Clara, CA, 1993: 534-537.
- [16] Mishchenko A. ABC: System for Sequential Logic Synthesis and Formal Verification[DB/OL]. [2021-12-15]. <https://github.com/berkeley-abc/abc>.
- [17] Yang S. Logic synthesis and optimization benchmarks user guide version 3.0[EB/OL]. [2022-01-09]. <https://max.book118.com/html/2017/0724/123971096.shtml>.

AIG based delay optimization of multilevel logic circuits using approximate computing technique

ZHAO Weikai, YU Zongyuan, WANG Lunyao^{*}

(Faculty of Electrical Engineering and Computer Science, Ningbo University, Ningbo 315211, China)

Abstract: In order to optimize the delay of multi-level circuits, an algorithm using the approximate replacement of the output node in critical path is proposed for delay optimization purposes. In the algorithm, those critical paths which can be optimized are searched first. Then, the nodes in each critical path with the lowest error rate are selected to form an approximate replacement node set. Finally, by deleting those nodes using approximate replacement under the constraint of error rate, the critical paths are shortened, and the delay is improved. The proposed algorithm is programmed with C++, built-in commands in ABC tools and tested with ISCAS85 and LGSynth91 benchmarks. The experimental results show that compared with the proposed constant replacement methods, the area and delay are improved by 22.96% and 31.49% respectively. Meanwhile, compared with the newly proposed algorithm, our algorithm running timing is improved by 61.88% while the optimization of delay and area remains similar.

Key words: approximate calculation; logic optimization; error constraint; multi-level circuit; delay optimization

(责任编辑 韩 超)