Graph-Oriented Layout Design for Field-coupled Nanocomputing via Parallel Multi-Objective Search Space Exploration

Simon Hofmann *Graduate Student Member, IEEE,* Marcel Walter *Member, IEEE,* and Robert Wille *Senior Member, IEEE*

***Abstract*—*Field-coupled Nanocomputing* (FCN) is a post-CMOS paradigm in which information propagates through near-field interactions rather than charge flow, enabling ultra-low-power, high-density logic. Translating netlists into manufacturable, cell-level layouts therefore becomes a pivotal challenge. Existing FCN physical design tools optimize only a single cost metric, typically footprint or runtime. As a result, designers must choose between exponentially slow exact solvers and fast yet area-intensive heuristics. We present the first FCN physical design engine that closes this gap by introducing configurable *effort modes*. These modes let users trade runtime for solution quality while simultaneously optimizing any discretionary objective, e. g. area, wire segments, crossings, or delay, thereby integrating data from physical simulation and manufacturing constraints. Our open-source implementation, released as part of the *Munich Nanotech Toolkit*, generates layouts for circuits that defeat state-of-the-art exact solvers. On such benchmarks, it shrinks footprint by an average of $73.07\,\%$, reduces crossings by $19.10\,\%$, and cuts wire segments by $54.47\,\%$ relative to a leading heuristic baseline. Even after post-layout optimization of the baseline, our approach still achieves mean gains of $25.99\,\%$ in area, $37.82\,\%$ in crossings, and $25.96\,\%$ in wire segments. These results establish the proposed engine as a compelling solution for highly optimized, large-scale standard-cell FCN design.**

***Index Terms*—Physical design, quantum cellular automata, quantum dots.**

I. Introduction

FIELD-COUPLED Nanocomputing (FCN, [1]) comprises a family of post-CMOS paradigms that exploit near-field interactions, rather than electric current, to transmit information. This approach enables logic at nanoscales while promising drastic reductions in energy consumption. Over the past few years, progress on four fronts has markedly accelerated FCN research:

1) **Device fabrication.** Advances in STM-based hydrogen lithography now allow the precise creation of functional elements [2] based on *Silicon Dangling Bonds* (SiDBs, [3]).
2) **Modeling and simulation.** SiDB logic can now be conceived and simulated efficiently using advanced gate designers and simulation engines [4]–[11].
3) **Physical design.** Dedicated placement, routing, and optimization flows translate gate-level netlists into FCN layouts, minimizing area, latency, and wiring overhead [12]–[29].
4) **Software tools.** For every part of the design flow, software tools have been developed, ranging from logic synthesis to physical design and simulation [5], [30]–[36].

Together, these advances position FCN, with the SiDB technology implementation in particular, as a credible contender for ultra-low-power, high-density computing in the post-CMOS era.

One of the four fronts is especially important for the success of FCN: physical design, which must translate a gate-level netlist into a layout, a process involving gate placement, wire routing, and signal clocking, while coping with tight technology constraints inherent to FCN, including clocking, signal synchronization, and limited crossing capabilities. Existing algorithms fall into two disjoint categories: exact approaches that guarantee optimal solutions at the cost of exponential runtimes, and scalable heuristics that are fast but sacrifice significant layout area. As a result, these algorithms give the designer no possibility to trade off runtime and solution quality. Historically, both classes of algorithms have pursued one single objective, namely the area footprint, as the layout size directly determines manufacturing cost and, in physical simulators, dominates execution time.

However, recent findings call for richer cost models. Physical simulation shows that gate robustness to external disturbances can vary widely across different logic gates [37]. Wire segments, in contrast to traditional CMOS, consume the same area as logic gates and furthermore introduce the same propagation delay [17], [38], while wire crossings severely complicate fabrication [39]. Moreover, multi-objective metrics, such as the area-delay or area-crossing products are largely unsupported.

Current algorithms cannot simultaneously optimize for such composite, technology-aware targets, nor can they scale effort according to design-time constraints. Consequently, physical design methods for FCN that can bridge the gap between exact and heuristic methods by offering graduated effort modes, and extend their optimization engines to support multi-objective cost functions that capture robustness, interconnect overheads, and number of crossings alongside area are desperately needed.

Simon Hofmann, Marcel Walter, and Robert Wille are with the Chair for Design Automation, Technical University of Munich, Munich, Germany. Simon Hofmann, Marcel Walter, and Robert Wille are also with the Munich Quantum Software Company GmbH, Garching near Munich, Germany. Robert Wille is also with the Software Competence Center Hagenberg GmbH, Hagenberg, Austria. E-mail: {simon.t.hofmann, marcel.walter, robert.wille}@tum.de

Additionally, digital and analog CMOS physical-design flows that balance area, timing, and power—whether by a single weighted cost function with hand-tuned coefficients [40] or Pareto-front sweeps that enumerate many trade-off points [41]—cannot be applied directly to FCN, whose distinctive device rules and clocking constraints require fundamentally different physical design and optimization strategies.

We present the first FCN physical design engine[1] that not only lets designers dial runtime versus solution quality through tunable *effort modes*, but also optimizes an arbitrary, user-defined mix of cost objectives, spanning, e. g., footprint, number of wire segments, number of crossings, or any weighted combination thereof, enabling the integration of insights from both simulation and manufacturing processes into physical design. Compared to a state-of-the-art heuristic baseline on benchmarks not solvable by exact approaches, the proposed algorithm reduces layout area, number of crossings, and number of wire segments by 73.07 %, 19.10 %, and 54.47 %, respectively.

An open-source implementation on top of the *fiction* framework [34] is available as part of the *Munich Nanotech Toolkit* (MNT, [42])[2] and also included in the co-design tool *MNT Designer*.[3] Furthermore, the generated layouts have been included in the benchmark suite *MNT Bench* [43],[4] replacing the latest best known solutions for multiple benchmark circuits.

The remainder of this paper is structured as follows: Section II reviews technical background on selected FCN technologies, as well as state-of-the-art physical design algorithms. Section III outlines the proposed efficient and scalable physical design algorithm with discretionary cost objectives and tunable effort modes, which is then experimentally evaluated in Section IV. Finally, Section V concludes the paper.

II. Background

This section first introduces the fundamentals of the two most prominent FCN technologies, namely *Quantum-dot Cellular Automata* (QCA, [44]) and *Silicon Dangling Bonds* (SiDBs, [3]). Afterward, the physical design problem in FCN is presented together with technological constraints inherent to most FCN implementations. Finally, current state-of-the-art approaches for FCN physical design are outlined.

A. Quantum-dot Cellular Automata (QCA)

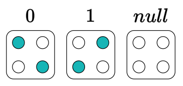
In QCA [30], [39], [44]–[46], the elementary building block is the *cell*: four quantum dots positioned at the corners of a square and jointly hosting two charges. Because the Coulomb repulsion along an edge is stronger than along a diagonal, the charges stabilize in one of two energetically favorable corner pairs. These charge configurations, depicted in Fig. 1a, encode the binary states 0 and 1. In an unexcited state, the charges are in superposition and therefore do not hold any information but only quantum noise, which can be defined as the *null* state, as seen in the rightmost QCA cell of Fig. 1a.

[1]Preliminary versions of this work have been published in [13], [23].

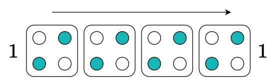
[2]Code: `https://github.com/cda-tum/fiction`.

[3]Code: `https://github.com/cda-tum/mnt-designer`.

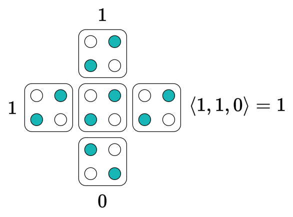
[4]`https://www.cda.cit.tum.de/mntbench`.



(a) The 0, 1, and *null* state.

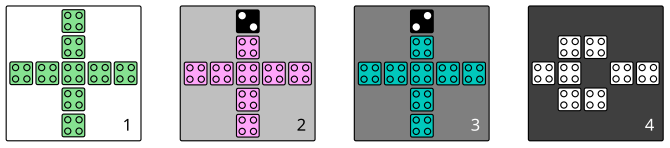


(b) Wire segment propagating the 1 state.

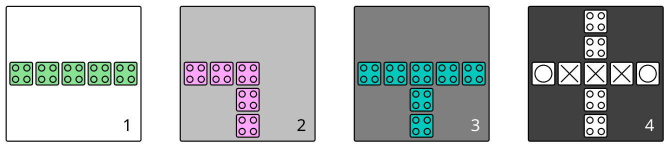


(c) Majority gate created by arranging five QCA cells.

Fig. 1: Elementary QCA cells and compound structures, adapted from [23].



(a) MAJ3 (b) AND (c) OR (d) Inverter



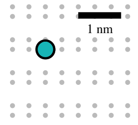
(e) Straight wire (f) Bent wire (g) Fan-out (h) Crossing

Fig. 2: Standard tiles in the QCA ONE gate library [47], taken from [23].

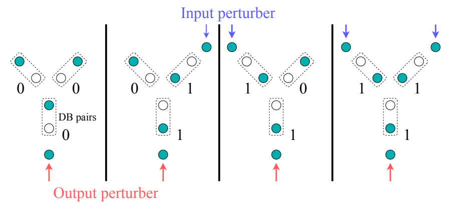
When cells are placed in close proximity, the electrostatic polarization of one cell biases its neighbor, producing a wire segment that transmits information, as illustrated in Fig. 1b. Additionally, the majority-of-three (MAJ3) function can be created by arranging five QCA cells as seen in Fig. 1c, where the QCA cells at the top, left and bottom all influence the cell in the middle. The state of the cell in the middle is therefore the majority of the three other cells' states. Based on this concept, multiple other gates can be created, like AND, OR, inverters or crossings, to build complete QCA gate libraries [47] as seen in Fig. 2.

B. Silicon Dangling Bonds (SiDBs)

SiDB technology replaces the four-dot QCA cell with a two-dot element to realize the *Binary-dot Logic* (BDL, [48]) concept. Each quantum dot is an individual SiDB created on a hydrogen-passivated silicon (H-Si(100)-2$\times$1) surface: a *Scanning-Tunneling-Microscope* (STM) tip selectively desorbs a single hydrogen atom, leaving behind an atomically-sized, chemically identical quantum dot that traps a localized charge [3], [49]. A schematic representation of the H-Si(100)-2$\times$1 surface with one present quantum dot can be seen in Fig. 3a. Advances in STM-based hydrogen lithography now allow ultimate precision in placing these dots [2], [50]–[53]. Exploiting this accuracy, researchers have fabricated a fully operational SiDB OR gate with a footprint of less than 30 nm^2 [48], which, together with other blueprints for

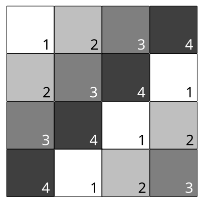


(a) H-Si(100)-2×1 surface structure.

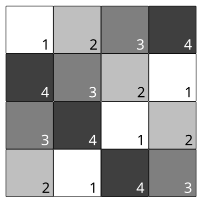


(b) Recreation of a binary-dot OR gate [48], adapted from [54].

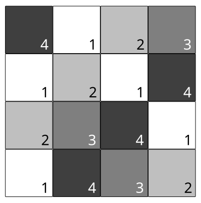
Fig. 3: SiDBs on an H-Si(100)-2×1 lattice implementing logic, taken from [28].



(a) *2DDWave* [57].



(b) *USE* [58].



(c) *RES* [59].

Fig. 5: Common clocking schemes for FCN technologies. The four distinct clock phases, labeled 1 through 4, are represented by white, light gray, medium gray, and dark gray, respectively, taken from [28].

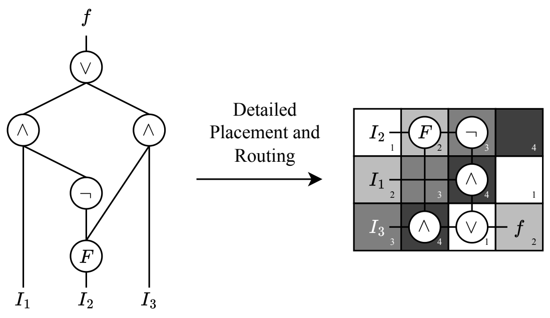


Fig. 4: A 2:1 Multiplexer is placed and routed on a layout.

basic gates, form the *Bestagon* gate library [54]. This OR gate is recreated in Fig. 3b, showing the four input combinations and the resulting output state, which is 0 if both inputs are 0 and 1 otherwise.

C. Physical Design Algorithms

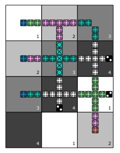
Physical design algorithms are essential for combining standard FCN building blocks into layouts that realize logic functions, yet they must operate within the stringent technological constraints of FCN devices. One example is shown in Fig. 4, where each logic gate in the network is placed on a layout and connected with its incoming and outgoing signals. The remainder of the section focuses on the technological constraints inherent to FCN physical design and an overview of current physical design algorithms.

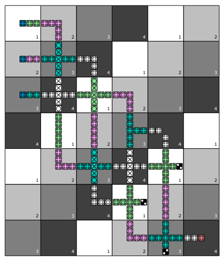
1) Technology Constraints: Wire routing is especially challenging: most FCN implementations are effectively planar and provide only limited crossing capabilities [39], [55], as well as the requirement to provide the same area for a wire segment compared to any other standard gate, as demonstrated by the gate library illustrated in Fig. 2. In addition, signal synchronization requires tight control of interconnect path lengths across the entire layout [56].

FCN circuits are partitioned into a grid of identical *tiles*, shown by the black borders surrounding each gate and interconnect segment in Fig. 2. The layout footprint is therefore measured simply by counting tiles. Reliable signal propagation is ensured through a four-phase clocking scheme (phases 1–4), with the proposed idea of buried electrodes in the substrate delivering these phase signals to every tile [45]. To facilitate the physical design process, different clocking schemes have been proposed, which offer tailored arrangements of regular clock zones. The most prominent clocking schemes are illustrated in Fig. 5, which are *2DDWave* [57] in Fig. 5a, *USE* [58] in Fig. 5b, and *RES* [59] in Fig. 5c. Recently, *2DDWave* has established itself as the superior clocking scheme, as information flow is restricted from left to right and top to bottom only, facilitating physical design [13]–[15], [17], [23], [28]. Furthermore, physical design algorithms agnostic to the clocking scheme usually create layouts with smaller footprint for *2DDWave* compared to other clocking schemes [12], [21], [25], [29].

Satisfying clock-phase synchronization, suppressing wire crossings and segment counts, and shrinking the layout footprint constitute a tightly coupled optimization problem. Even the seemingly simpler task of minimizing area alone is $\mathcal{NP}$-complete [60]. Unlike CMOS place-and-route flows, where an entire standard cell is the fundamental unit, FCN physical design must first locate every individual logic gate and only afterwards substitute it with the corresponding multi-dot implementation drawn from a chosen library [47], [54]. Hence, a practical FCN layout generation algorithm must respect all technology-specific wiring and clocking constraints while simultaneously optimizing whichever cost objectives the designer prioritizes. Moreover, a practical physical design approach must simultaneously balance layout quality and computational runtime, yet prior algorithms have generally optimized one at the expense of the other.

2) Exact Approaches: Several works [12], [29] have cast the FCN physical design problem as a fully symbolic formulation and solved it with SMT-based reasoning engines. These methods enumerate candidate grid sizes in ascending order and, for each size, ask the solver whether a placement and routing that realizes the target Boolean function exists. When more than one solution is feasible for a given area, additional solver constraints allow secondary objectives (e.g., number of wire segments or number of crossings) to be imposed. Although provably optimal, these techniques inherit the $\mathcal{NP}$-completeness of FCN physical design [60] and have thus far been practical only for networks of roughly 40 gates or less.


(a) Created by *exact* [12].


(b) Created by *ortho* [14].

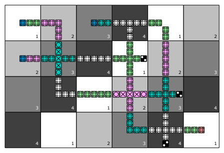

(c) Created by *ortho* with subsequent *PLO* [28].

Fig. 6: Layouts implementing the 2:1 multiplexer created by three different physical design algorithms.

Example 1. *One example for an optimal placement and routing is shown in Fig. 6a, where a 2:1 multiplexer is realized using only twelve tiles.*

3) Heuristic Approaches: Since exact approaches only scale up to around 40 gates, a range of heuristic strategies have been proposed that trade optimality for tractability. These methods either prioritize raw scalability, often ignoring area overhead altogether, or seek smaller areas while relying on strong search space restrictions.

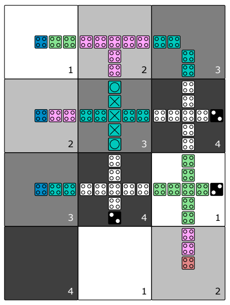
The algorithms *ortho* [14] and its input-ordering extension (*IO-SDN*) [16] illustrate the first class: by framing the placement and routing problem as orthogonal graph drawing on a Cartesian grid, they lay out QCA circuits with hundreds of gates in milliseconds, but the resulting area can be an order of magnitude larger than the optimum. Additionally, due to its reliance on signal flow directions to be restricted from top to bottom and left to right, only the *2DDWave* clocking scheme can be used.

Example 2. *For the 2:1 multiplexer shown in Fig. 4, the resulting layout has* 42 *tiles,* 30 *tiles more compared to the optimum.*

Another approach applies reinforcement learning to gate placement [21], [25]. The agent receives a reward shaped by the current layout area and gradually learns placement policies that outperform hand-crafted heuristics, yet they still fall short of the provably optimal layouts obtainable by exact solvers.

4) Post-Layout Optimization: To mitigate the area overhead produced by heuristic placers, a separate post-layout optimization phase can be applied [15], [17], [28]. After a first legal placement is fixed, the optimizer first relocates selected gates to better positions and then removes excess wiring, thereby freeing up unused rows and columns. In practice this optimization step recovers a substantial fraction of the area lost during the initial heuristic design.

Example 3. *For the 2:1 multiplexer created by* ortho *shown in Fig. 6b, post-layout optimization is able to reduce the layout area by* 18 *tiles, resulting in a layout with* 24 *tiles shown in Fig. 6c. Notably, this layout is still twice as large as the optimal one in Fig. 6a.*


(a) Cartesian layout with QCA gates from the QCA ONE gate library [47].

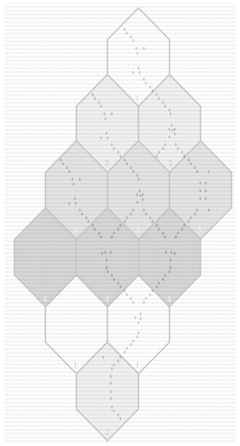

(b) Hexagonal layout with SiDB gates from the *Bestagon* gate library [54].

Fig. 7: Cartesian layouts can be transformed into hexagonal layouts using a 45° turn [19].

5) Hexagonalization: To transform any Cartesian, *2DDWave*-clocked layout into a hexagonal configuration to accommodate Y-shaped SiDB gates, an algorithm [19] utilizing a 45° turn can be used.

Example 4. *The optimal implementation in Fig. 7a of the 2:1 multiplexer using QCA gates from QCA ONE [47] is transformed into its hexagonal representation in Fig. 7b using SiDB gates from the* Bestagon *library [54] by rotating each tile and stretching it vertically.*

Therefore, 2DDWave is selected as the underlying clocking scheme for the proposed algorithm, since it enables the strongest search space restrictions, and any resulting Cartesian layout suitable for QCA can later be mapped to a hexagonal form for Y-shaped SiDB gates, making it technology-agnostic.

III. Efficient and Scalable Layout Design with Discretionary Cost Objectives

In this section, we first present the general idea and an overview of the proposed algorithm for designing FCN gate-level layouts in Section III-A and Section III-B, and then describe the creation of multiple *search space graphs* (SSGs) in Section III-C. Afterward, we detail the main search procedure in Section III-D, and finally explain how different cost objectives are included in Section III-E.

A. General Idea

The main idea is to frame physical design as a path-finding problem on an SSG. Gates are placed one by one in a topological ordering, where each placement decision generates a new vertex that captures the current partial layout. Two vertices are joined by an edge when the latter can be obtained from the former by inserting exactly one additional gate. While this incremental placement proceeds, an *A*-Search algorithm* [61] is run not only to decide which vertex to expand next based

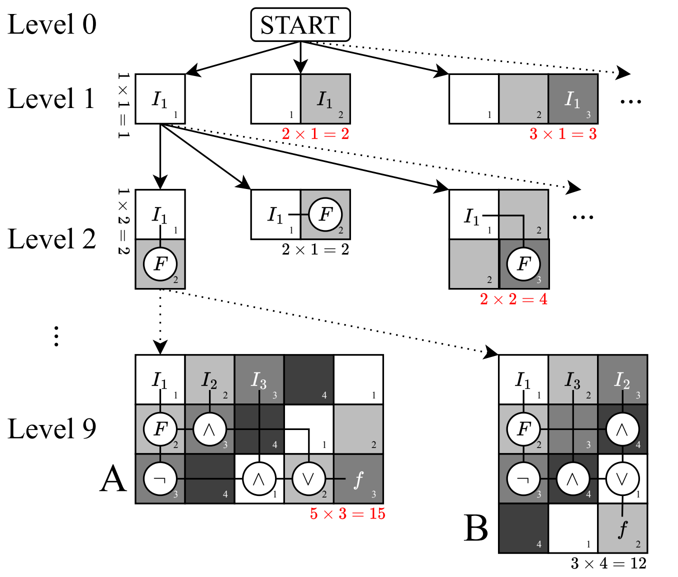


Fig. 8: The search space graph created for the 2:1 multiplexer, with two possible solutions at the bottom, taken from [23].

on the desired cost objective but also to carry out routing for the already placed subset.

The framework is objective-agnostic, any metric (e. g., layout area or number of crossings) can steer the search, and inherently parallel: independent graphs can be spawned to explore alternative global choices such as input pin locations or fanout tree creation. An effort knob determines how many graphs are explored concurrently, allowing designers to trade off runtime against layout quality.

Example 5. *Fig. 8 shows the SSG exploration for a 2:1 multiplexer. The root (level 0) is empty; level 1 contains one vertex for every legal placement of the first primary input pin along the top row. At each subsequent level the algorithm expands the vertex whose cost—here equal to the current footprint in tiles—is minimal, so the search always pursues the most compact partial layout first.*

When the last level is reached, every vertex corresponds to a complete, functionally correct layout. Two candidates, labeled A *and* B*, are highlighted: both implement the multiplexer, but* B *occupies fewer tiles than* A*. Although* A *is discovered first, the algorithm backtracks within the SSG, reopens a higher-cost vertex, and ultimately finds* B*. Because* B *has the lowest cost at the terminal level, it is returned as the final solution.*

B. Algorithm Overview

The *graph-oriented layout design* (gold) algorithm, as outlined in Algorithm 1, constructs gate-level layouts by exploring one or more SSGs, each of whose vertices encode a *partial layout*. The input *network* is represented purely in terms of gates and interconnecting signals, with no gate-level placement or routing information. A *partial layout* is a legal, partially routed embedding in which exactly k of the N gates have been placed, and all interconnecting signals between the placed gates have been routed.

Algorithm 1: Graph-Oriented Layout Design

```
Function GraphOrientedLayoutDesign(network):
    $\mathcal{S} \leftarrow$ INITIALIZESSGS($network$, $effortMode$)
    $bestLayout \leftarrow \varnothing$
    while $\exists\, ssg \in \mathcal{S} : ssg.active$ do
        parallel for $ssg \in \mathcal{S}$ where $ssg.active$ do
            $L \leftarrow$ Expand($ssg$)
            if $L \neq \varnothing$ and BETTER($L$, $bestLayout$) then
                $bestLayout \leftarrow L$
            end
            if $ssg.frontier$.EMPTY() then
                $ssg.active \leftarrow$ false
            else
                $ssg.current \leftarrow ssg.frontier$.POPMIN()
            end
        end
    end
    return $bestLayout$
end

Function Expand(ssg):
    $layout \leftarrow$ GETPARTIALLAYOUT($ssg.current$)
    if INVALID($layout$) then
        return $\varnothing$
    end
    if ALLPLACED($layout$) then
        return $layout$
    end
    foreach $p \in$ FEASIBLETILES($layout$, $ssg$, $numExpansions$) do
        $v \leftarrow ssg.current \,\|\, p$
        $g \leftarrow ssg.numRemainingNodes$
        $h \leftarrow$ HEURISTIC($layout$, $p$, $ssg.costObjective$)
        $f \leftarrow g + h$
        $ssg.frontier$.PUSH($v$, $f$)
    end
    return $\varnothing$
end
```

Each vertex in an SSG is identified by an ordered tuple

$$v \;=\; \big\langle (x_0, y_0), (x_1, y_1),\, \ldots,\, (x_{k-1}, y_{k-1}) \big\rangle,$$

where (x_i, y_i) is the grid coordinate (tile) chosen for the i^{th} gate in a fixed topological ordering. To guide the best-first expansion of these vertices, *gold* maintains a *frontier queue*, which is a min-priority queue keyed by

$$f(v) \;=\; g(v) \;+\; h(v)\,.$$

Here, $g(v)$ is simply the number of gates remaining to place (i. e., $g(v) = N - k$), ensuring that the successful placement of gates is preferred to minimizing other cost objectives such as the number of wire segments. The term $h(v)$ is a normalized cost heuristic on the *current* partial layout—such as the occupied area, number of wire segments, crossing count, or a user-defined combination, scaled into the interval $[0, 1)$ to bias the expansion toward low-cost embeddings.

To achieve this, the algorithm carries out the following steps:

1. **Create alternative search spaces.** To avoid the intractability of exploring the full search space, it is partitioned into a family $\mathcal{S}$ of SSGs (Line 2). Each SSG is obtained by fixing exactly one choice along each *algorithmic dimension* (primary input placement, fanout substitution strategy, and gate ordering), thereby yielding smaller, more manageable search spaces that collectively cover all combinations of

these dimensions. The number of generated search space graphs is determined by the *effort mode*.

2. **Maintain the best layout.** The variable *bestLayout* is initialized to an empty layout (Line 3) and is updated whenever a superior complete layout is discovered.
3. **Parallel search loop.** While at least one SSG remains active (Line 4), the algorithm processes all active SSGs in parallel (Line 5) by
   a) expanding each SSG's frontier to generate new partial layouts (Line 6);
   b) checking if a newly generated *complete* layout is better than the current best solution (Line 7) to replace *bestLayout* (Line 8);
   c) deactivating exhausted SSGs whose frontier becomes empty (Line 10) by marking them as inactive (Line 11).
4. **Expansion within one SSG.** When expanding a single SSG, the algorithm repeatedly
   a) removes the vertex v with the smallest cost $f(v)$ (Line 13);
   b) reconstructs the partial layout for v (Line 21);
   c) checks if the layout is infeasible (Line 22), i. e., the partial layout is impossible to complete due to trapped gates or unavailable routing paths, in which case an empty layout is returned (Line 23);
   d) checks if the layout is already complete (Line 25), and, if so, returns it (Line 26);
   e) otherwise, for a predefined number of feasible tiles p for the next gate (Line 28),
      - form the successor by appending p to v: $v' = v \,\|\, p$ (Line 29),
      - compute its cost $f(v') = g(v') + h(v')$ (Line 30 to Line 32),
      - and insert v' into the frontier (Line 33).
5. **Termination.** When all SSGs are inactive or a set timeout is reached, the algorithm returns the best complete layout obtained so far (Line 17).

Because the SSGs differ only in high-level policies (primary input placement, fanout substitution strategy, topological gate order), they can be explored independently yet compared fairly based on the specified cost objective. Parallel exploration increases the chance that at least one SSG yields a complete layout before the timeout is reached.

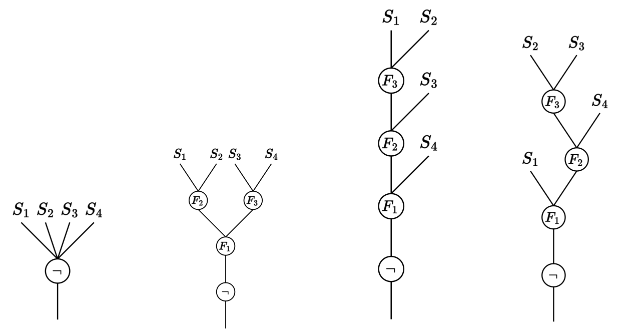
C. Algorithmic Dimensions

To manage the vast search space, each SSG is restricted by high-level algorithmic dimensions that only influence the placement location of primary inputs and fanouts or the topological ordering in which the gates are placed, but not its functionality, number of gates, or choice of logic gates.

The algorithmic dimensions used to instantiate different SSGs are as follows:

1) PI Placement Policy: Primary inputs (PIs) are seeded along the boundaries of the layout in one of the following ways:

- all PIs placed in the topmost row;
- all PIs placed in the leftmost column;



(a) Input network. (b) Breadth-first. (c) Depth-first. (d) Random.

Fig. 9: Three different fanout substitution strategies used to create SSGs in *gold*.

- PIs may occupy either the top row or left column.

2) Fanout Substitution Strategy: As outlined in the preliminaries, gates in FCN can only have one outgoing signal and splitting of signals can only be achieved using dedicated fanout elements. Additionally, 2DDWave-clocked layouts restrict each fanout to at most two outgoing signals. Consequently, whenever a gate has multiple outgoing connections, we leave the gate itself intact and substitute its outgoing signals with a fanout tree composed of one or more fanouts.

Three strategies are used:

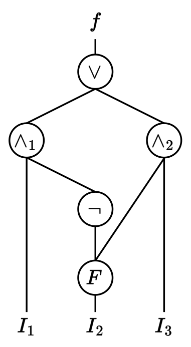
- insert fanouts breadth-first, minimizing tree depth;
- insert fanouts depth-first, minimizing tree width;
- randomize fanout insertion order.

Example 6. *In Fig. 9, the three different fanout substitution strategies are illustrated by means of the trivial network in Fig. 9a, which consists of a single inverter $\neg$ and four outgoing signals $S_1 \dots S_4$.*

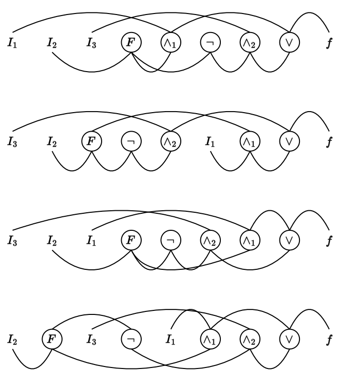
- ***Breadth-first*** *(Fig. 9b). After the first fanout F_1 splitting up the outgoing signals, two more fanouts F_2 and F_3 are inserted, each driving two signals. The resulting tree has a depth of 2 and a perfectly balanced shape with a width of 2, minimizing the longest signal path.*
- ***Depth-first*** *(Fig. 9c). A single fanout F_1 is attached to the inverter and drives S_4 as well as a second fanout F_2, which drives S_3 and the third fanout F_3. Finally, the last fanout F_3 then splits up the signal to S_1 and S_2. This produces a tree of depth 3 and width of 1.*
- ***Random order*** *(Fig. 9d). Fanouts are placed in an arbitrary sequence; in the depicted run, the fanout tree looks similar to the depth-first one, but the order of the outgoing signals is different.*

3) Topological Node Ordering: To route a newly placed gate with its predecessors, they must already be present in the layout, which is ensured by placing gates in a topological order. Since the chosen ordering affects routing congestion and the overall layout quality, multiple topological orderings are generated according to four different traversal strategies:

- traverse from outputs backward to inputs (PO→PI);
- traverse from inputs forward to outputs (PI→PO);



(a) Input network.



(b) Four topological orderings.

Fig. 10: Buffered network and different topological orderings.

Table I: SSGs per effort mode. *Factorization* shows the option counts for each dimension.

Effort Mode	Total SSGs	Factorization[†]
High-Efficiency	2	$1 \times 1 \times 2 \times 1 \times 1$
High-Effort	12	$3 \times 2 \times 2 \times 1 \times 1$
Highest-Effort	48	$3 \times 2 \times 2 \times 4 \times 1$
Maximum-Effort	96	$3 \times 2 \times 2 \times 4 \times 2$

[†]Counts are listed in the order: PI placement policy, fanout substitution strategy, topological ordering, cost objective (layout area, number of wires, number of crossings, or area-crossing product), and extra randomization.

- traverse PO→PI with randomized fanin exploration;
- traverse PI→PO with randomized fanout exploration.

Example 7. *Fig. 10b illustrates four valid topological orderings based on the four aforementioned substitution strategies, from top to bottom, for the buffered network in Fig. 10a. All respect topological precedence because for each gate, all incoming gates and primary inputs are placed to the left.*

4) SSG Count by Effort Mode: To trade off runtime efficiency and result quality, the number of generated SSGs is scaled during instantiation using the *effortMode* parameter (Line 2). Table I summarizes how many SSGs are instantiated for each effort mode using the option counts shown in the right-most column. Each SSG is defined by one choice from three algorithmic dimensions (PI placement policy, fanout substitution strategy, topological ordering), and by one of several cost objectives. For HIGH-EFFICIENCY, HIGH-EFFORT, and HIGHEST-EFFORT, the fanout substitution strategies and topological orderings only include the deterministic, not randomized variants, while for the MAXIMUM-EFFORT mode, the number of SSGs is double compared to HIGHEST-EFFORT using randomization.

D. Search Procedure

This section further explains how a vertex in the SSG gets extended, i. e., how a partial layout with k gates placed gets transformed into a layout with $k + 1$ gates placed.

The outer **while** loop (Line 4) terminates when every SSG has become inactive or a user-defined timeout occurs. Each active SSG is processed by a separate worker thread (Line 5 to Line 14). The only shared resource is the global best layout.

For a given SSG, `Expand()` (Line 6) performs three tasks:

1) **Materialize partial layout.** The vertex stored in *current* is replayed to obtain the corresponding partial layout (Line 21).
2) **Validate and finish.** If the layout is invalid, it is discarded immediately (Line 22). If all N gates are placed (Line 25), the fully routed layout is returned.
3) **Generate successors.** Otherwise, up to *numExpansions* feasible tiles for the next logic node are computed. Every tile p yields a successor vertex $v' = v \,\|\, p$ together with its key $f(v')$ (Line 28 to Line 33).

The successors are then added to the frontier (Line 7–Line 13) and the partial layout with the lowest cost becomes the new *current* layout to expand further.

Successor tiles depend on the type of the next node:

- **PI.** A tile on the chosen boundary that still sees a possible path to either the right or bottom border.
- **PO.** A tile on the right or bottom border that sees a possible path from its predecessor.
- **Single-fanin gate.** A tile that (i) sees a possible path from its predecessor, and (ii) leaves at least one escape path towards the right or bottom border.
- **Dual-fanin gate.** A tile that (i) sees a possible path from each of its predecessors, and (ii) leaves at least one escape path towards the right or bottom border.
- **Dual-fanout gate.** A tile that (i) sees a possible path from its predecessor, and (ii) leaves at least two escape paths towards the right or bottom border.

Each condition is checked using an A^* path-finding algorithm.

When a solution is found, *gold* revisits vertices with a higher cost in terms of placed gates ($g(v)$) to discover solutions with a lower cost in terms of layout area, number of crossings, or any other discretionary cost objective ($h(v)$), which we denote as *backtracking* through the SSG.

The cost of the best globally found layout can be used to prune other SSGs by discarding all vertices that already have a higher cost in terms of, e. g., layout area or number of crossings, as these costs only increase when placing more nodes in the layout. These pruning strategies can be classified into two rules:

1) **Cost pruning.** If the partial layout associated with a vertex already has a higher desired cost ($h(v) \geq h(best)$), the partial layout is skipped and marked as not worth expanding further.
2) **Objective pruning.** When the SSGs contain several cost objectives, the cost of the best layout in terms of the desired cost function (e. g., layout area or number of crossings) is used to prune all SSGs independent of the cost function used in them.

E. Cost Objectives

In addition to single optimization objectives like layout area, number of crossings, or number of wire segments, *gold* also supports composite objectives familiar from CMOS physical

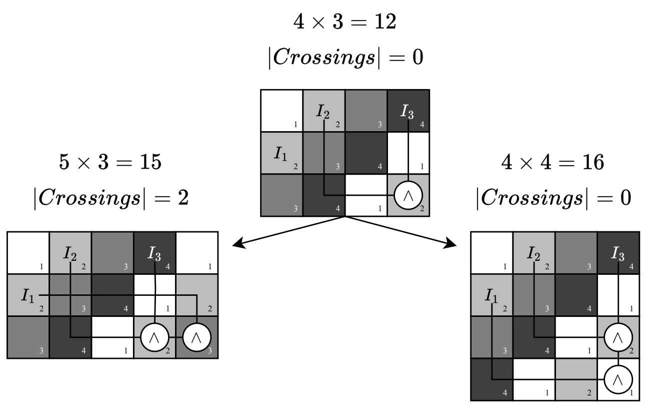


Fig. 11: Depending on the cost objective, different expansions are prioritized, taken from [23].

design, such as the *Area-Delay Product* (ADP) and *Power-Delay Product* (PDP) [62].

In FCN, crossings are particularly sensitive and difficult to fabricate, a possible cost metric to trade off the number of crossings and the layout area is therefore the *Area-Crossing Product* (ACP):

$$ACP = A \cdot (|C| + 1), \tag{1}$$

where A denotes the current layout area and $|C|$ the number of wire crossings. Moreover, users can specify *any* cost objective computable from the current partial layout. Because this objective must be evaluated at every expansion step, its computation time directly influences the overall scalability.

The total cost is defined as

$$f(v) = \underbrace{(N - |v|)}_{\text{gates still missing}} + \underbrace{\frac{\theta(\text{layout}(v))}{K}}_{\text{defined cost}},$$

where $N - |v|$ is the number of gates that remain to be placed, while $\theta(\text{layout}(v))$ represents an arbitrary cost metric, for example the current layout area, evaluated on the partial layout associated with vertex v. Because this user-defined metric serves only as a *secondary* optimization criterion, it is rescaled to the interval $[0, 1)$ by dividing it by a constant K that is strictly greater than any value θ can attain.

Example 8. *In Fig. 11, the top row shows a partial layout with three input pins and a single placed gate. Where the next* AND *gate is inserted depends on the chosen cost objective:*

- ***Area-driven placement*** *Placing the gate to the* right *of the existing one (lower-left layout) minimizes the footprint to* $5 \times 3 = 15$ *tiles, but forces the wire from* I_1 *to cross two other wires, adding two crossings.*
- ***Crossing-driven placement*** *Placing the gate* below *the existing one (lower-right layout) avoids introducing any additional crossings, yet enlarges the footprint to* $4 \times 4 = 16$ *tiles.*

This example illustrates the inherent trade-off: reducing wire crossings can inflate area, and vice versa.

IV. Experimental Evaluation

To evaluate the practical performance of the proposed physical design algorithm *gold*, we carried out a comprehensive experimental study benchmarking it against both exact and heuristic state-of-the-art methods. After detailing the experimental setup in Section IV-A, we present the numerical results in Section IV-B and Section IV-C. Finally, Section IV-D presents three illustrative examples, and Section IV-E offers a discussion of the results along with an outlook.

A. Experimental Setup

In total, 35 combinational circuits of widely varying size and structure, ranging from 9 to 151 gates, were taken from three public benchmark suites (Trindade16 [63], Fontes18 [64], and IWLS93 [65]). The parameter *numExpansions* was set to 4 if not stated otherwise. Solution quality is assessed by three complementary metrics:

- **Total area** (A) of the final gate-level layout (in tiles),
- **Number of wire crossings** ($|C|$), and
- **Number of wire segments** ($|W|$).

All experiments were executed under macOS 15.5 on an Apple-Silicon M1 Pro (8 performance + 2 efficiency cores at 3.20 GHz, 16 GB unified DRAM). Each layout attempt was capped at 60 s runtime; functional correctness of every solution was confirmed with the SAT-based equivalence checker proposed in [66].

Table II compares *gold* against three state-of-the-art baselines:

1) ***exact*** [12]. An SMT-based algorithm that guarantees global optimality w.r.t. area, but scales exponentially in terms of runtime based on the number of gates.
2) ***ortho*** [14]. A fast and scalable heuristic based on orthogonal graph-drawing; and
3) ***ortho + PLO*** [15]. The aforementioned heuristic post-processed with a gate relocation and wiring reduction post-layout optimization algorithm, denoted as *PLO*.

The proposed algorithm *gold* is invoked with the three different cost objectives, denoted as *gold*(X) with $X \in \{A, C, W\}$.

B. Numerical Results

Table II reports the complete results for all 35 benchmarks, where within each row the best figure is highlighted in bold. We first summarize the 17 small circuits solvable by *exact* and then discuss the 18 larger ones that can only be solved by heuristic approaches.

a) Instances ($|G| \leq 40$) solvable by exact (17/35): All 7 Trindade16 circuits and 10 of the Fontes18 circuits can be solved optimally by *exact* within the one-minute timeout limit. If multiple optimal layouts in terms of area have been found, the one with the lowest number of crossings and wire segments was chosen. On this small-to-medium subset

- *gold*(A) exactly matches the minimum area in **7** cases;
- *gold*(C) eliminates extra crossings in **5** and achieves the same number of crossings in **7**; and

Table II: Experimental evaluation of the proposed algorithm for different cost objectives.

Benchmark Circuit [63], [64]			exact			ortho			ortho + PLO			gold(A)			gold(C)			gold(W)		
	Name	$\|G\|$	A	$\|C\|$	$\|W\|$	A	$\|C\|$	$\|W\|$	A	$\|C\|$	$\|W\|$	A	$\|C\|$	$\|W\|$	A	$\|C\|$	$\|W\|$	A	$\|C\|$	$\|W\|$
Trindade16 [63]	2:1 MUX	9	**12**	1	3	42	5	20	24	2	11	**12**	1	3	15	**0**	5	15	**0**	**2**
	XOR	9	**18**	**1**	**7**	35	2	14	20	**1**	8	**18**	**1**	9	21	**1**	11	**18**	**1**	**7**
	XNOR	11	**18**	**1**	**4**	48	2	17	35	2	14	**18**	**1**	6	27	**1**	12	**18**	**1**	**4**
	Half Adder	14	**24**	**2**	**12**	72	5	35	**24**	**2**	13	**24**	4	14	40	**2**	23	**24**	4	14
	Full Adder	14	**28**	**2**	**15**	80	6	40	42	4	28	32	3	19	32	**2**	19	32	3	19
	Parity Check	15	**48**	4	**16**	228	22	102	88	12	50	50	4	22	60	**3**	27	54	5	21
	Parity Gen.	18	**32**	**2**	**8**	117	6	52	63	6	36	**32**	**2**	12	40	**2**	18	**32**	**2**	10
Fontes18 [64]	c17	18	**28**	**1**	**9**	130	16	65	90	15	51	**28**	**1**	11	36	**1**	17	30	**1**	11
	t	21	**30**	**1**	**9**	160	16	74	70	10	42	32	2	13	36	2	14	32	2	13
	t_5	21	**30**	**1**	**9**	160	15	75	42	4	19	**30**	**1**	11	**30**	**1**	11	**30**	**1**	11
	1bitAdderAOIG	26	**55**	**2**	**20**	216	14	88	130	15	63	65	5	36	65	3	35	65	3	35
	b1_r2	26	**40**	5	**20**	221	15	105	96	9	51	56	5	30	64	**3**	31	64	4	29
	majority	27	**48**	3	**14**	216	22	126	112	18	71	75	**2**	40	96	**2**	49	75	4	39
	majority_5_r1	27	**44**	**2**	**14**	230	21	123	126	17	74	54	5	26	54	4	26	54	5	25
	newtag	28	**44**	3	**15**	300	32	169	120	20	73	52	**1**	20	102	**1**	48	52	**1**	18
	clpl	30	**45**	**0**	**4**	425	77	247	120	21	90	70	8	39	120	6	69	70	8	39
	XOR5_R1	40	**77**	**4**	**20**	448	19	184	190	22	100	90	8	41	176	6	70	90	8	41
	1bitAdderMaj	45		—		490	33	199	364	42	191	**200**	20	**115**	238	**14**	118	**200**	20	**115**
	cm82a_5	68		—		1248	52	444	368	71	260	**234**	44	**176**	324	**40**	197	**234**	44	**176**
	2bitAdderMaj	82		—		1674	71	484	780	81	364	**374**	34	**184**	420	**33**	194	**374**	34	**184**
	xor5Maj	102		—		2418	113	809	1248	177	651	**726**	**70**	**370**	759	**70**	373	**726**	**70**	**370**
	parity	150		—		5712	164	1850	2028	267	1064	**504**	32	**246**	836	**25**	264	**504**	32	**246**
IWLS93 [65]	b1	53		—		740	46	344	234	43	178	**132**	35	**111**	180	**33**	122	**132**	35	**111**
	majority	60		—		920	76	439	288	59	213	**264**	46	187	360	**43**	227	273	44	**176**
	con1	66		—		1122	108	537	464	105	383	**264**	48	**202**	396	**43**	262	**264**	48	**202**
	cm138a	71		—		1485	63	547	315	54	**203**	**306**	72	235	374	**48**	239	323	53	208
	cm82a	74		—		1430	56	534	522	72	330	**336**	**38**	**189**	**336**	**38**	**189**	**336**	**38**	**189**
	cm42a	79		—		1739	**61**	683	**510**	93	**374**	704	142	513	704	142	513	704	142	513
	cm152a	86		—		2112	152	971	1232	269	896	**684**	189	589	910	**91**	549	756	115	**504**
	decod	107		—		3162	**119**	1358	**968**	227	**799**	2080	445	1725	2112	445	1738	2080	445	1725
	cm151a	122		—		3626	234	1594	1881	296	1011	**782**	98	510	**782**	**88**	**507**	**782**	**88**	**507**
	i1	125		—		5264	560	2621	1836	449	1496	**1287**	**325**	**1203**	**1287**	**325**	**1203**	**1287**	**325**	**1203**
	cm85a	132		—		4455	250	1872	1470	262	1011	**528**	**74**	**354**	598	**74**	362	**528**	**74**	**354**
	tcon	144		—		5985	339	2498	1488	390	1404	**1100**	197	973	1140	**183**	**964**	1140	**183**	**964**
	cmb	151		—		6272	516	2863	2537	596	2047	**2187**	**285**	**1502**	2214	**285**	1503	**2187**	**285**	**1502**

$|G|$ represents the total number of gates in the logic network, including primary inputs, primary outputs, and fanout buffers; A, $|C|$, and $|W|$ denote the resulting layout area, number of crossings, and number of wire segments, respectively. The timeout for each layout generation was set to $1\,\text{min}$.

Table III: Summary of *gold* unique wins and average improvements on the 18 large benchmarks ($|G| > 40$).

Objective	Unique Wins	Average Improvement	
	(out of 18)	vs. *ortho*	vs. *ortho + PLO*
gold(A)	16	73.07 %	25.99 %
gold(C)	16	19.10 %	37.82 %
gold(W)	15	54.47 %	25.96 %

Table IV: Peak improvement for the `parity` function.

Objective	Peak gain (vs. *ortho*)	Peak Gain (vs. *ortho + PLO*)
gold(A)	91.18 %	75.15 %
gold(C)	84.76 %	90.64 %
gold(W)	86.70 %	76.88 %

- *gold*(W) achieves the same number of wire segments in **3** relative to those area-optimal layouts.

Due to the layout area and number of wire segments being highly intertwined metrics, as well as *gold* relying on heuristic path-finding algorithms for routing, the slight superiority of *exact* was therefore expected. For other cost metrics, like the number of crossings, *gold* is able to find superior solutions compared to *exact*, as the layout with the least amount of crossings is not necessarily the one with the lowest area footprint, as eliminating a crossing might lead to routing detours.

b) Instances ($|G| > 40$) not solvable by exact (18/35): On the remaining 18 circuits, the advantage shifts clearly to *gold*, even against the strongest heuristic baseline *ortho* with *PLO*. Table III summarizes the average and peak improvements of *gold* against the two heuristic baselines. Notably, *gold*(A) achieves 73.07 % average area reduction compared to *ortho* and 25.99 % compared to *ortho + PLO*. In terms of the number of crossings, *gold*(C) achieves 19.10 % average crossing reduction compared to *ortho* and 37.82 % compared to *ortho + PLO*. For the number of wire segments, *gold*(W) achieves 54.47 % wiring reduction compared to *ortho* and 25.96 % compared to *ortho + PLO*. The highest improvement, as summarized in Table IV, was found for the second largest benchmark circuit, `parity`: 91.18 % (75.15 %) area reduction, 84.76 % (90.64 %) crossing reduction, and 86.70 % (76.88 %) wiring reduction compared to *ortho* (+ *PLO*).

c) Runtime vs. Gate Count: For all 35 benchmarks the first legal layout is found in Maximum-Effort mode within 1.5 s, rising from ≈0.02 s at 9 gates to ≈1.43 s at 151 gates, as shown in Fig. 12. The dashed quadratic fit merely captures this empirical range; it should not be extrapolated to substantially larger networks, where the number of partial placements in every SSG grows exponentially and backtracking becomes infeasible.

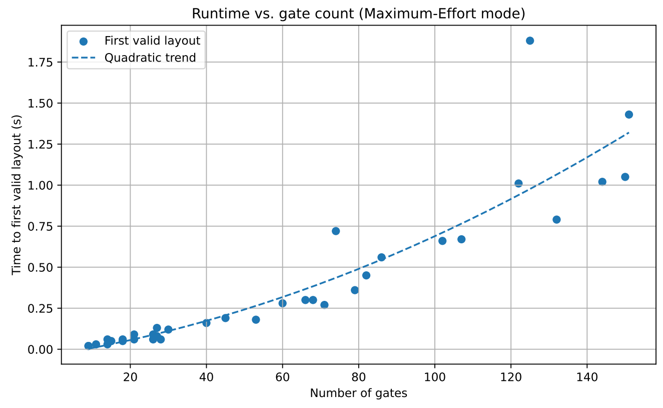
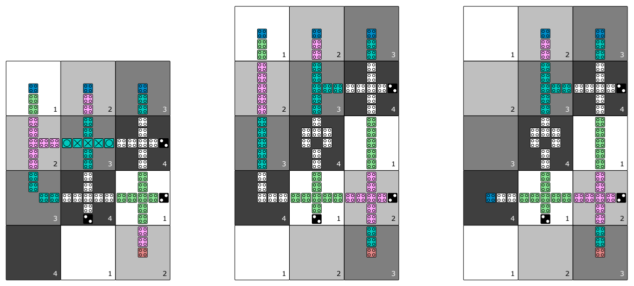


Fig. 12: Runtime to determine first valid layout in MAXIMUM-EFFORT mode.



(a) Layout with **12 tiles of area**, 1 crossing, and 3 wire segments.

(b) Layout with 15 tiles of area, **0 crossings**, and 5 wire segments.

(c) Layout with 15 tiles of area, **0 crossings**, and **2 wire segments**.

Fig. 13: Three layouts for the 2:1 MUX function based on the three different cost objectives area (A), number of crossings ($|C|$), and number of wire segments ($|W|$).

C. *Parameter Sweeps*

This section investigates how key algorithm parameters influence layout quality and runtime. We systematically sweep settings such as the effort mode, expansion width, and threading configuration. The goal is to guide users toward configurations that best match their design goals—whether fast legal solutions or highly compact layouts are required—under various resource constraints.

1) Effort-Mode Runtime/Quality Trade-off: Table V contrasts the four *effort modes*. Two observations suffice:

- **Instant first solutions.** The lightest HIGH-EFFICIENCY setting (2 SSGs) returns a legal layout in < 0.1 s for almost all benchmark circuits; ramping up to MAXIMUM-EFFORT still keeps the runtime below or close to 1 s.
- **Higher effort modes lead to better solutions.** Moving from HIGH-EFFICIENCY to MAXIMUM-EFFORT ($2 \rightarrow 96$ SSGs) drastically cuts the final layout area when setting the timeout limit to 100 s.

Therefore, a general recommendation for the designer is to use HIGH-EFFICIENCY (or MAXIMUM-EFFORT with a sub-second timeout) when a legal layout is needed almost immediately. When layout compactness matters more than turnaround, HIGH- to MAXIMUM-EFFORT should be selected and a longer timeout allowed.

2) Impact of numExpansions*:* The parameter *numExpansions*, which defines how many candidate tiles are explored at each expansion, directly trades runtime against layout quality and is evaluated in Table VI. Choosing a *small* value (1-2)

- keeps the time to find the first solution minimal and lets the search backtrack almost the entire SSG within a generous timeout, which works well for small or medium networks, but
- can lead to no valid solution being found at all, as for `xor5Maj` with *numExpansions* set to 1.

Raising *numExpansions* beyond 6 rarely shrinks the area further, yet it inflates runtime substantially. In practice the default is set to **4** in the algorithm with the recommendation to stay in the range **2-6** and to increase the timeout rather than the expansion width when additional quality is needed.

3) Single- vs. Multithreaded Performance: Multithreading with 10 cores leads to consistent performance improvements across all benchmarks, with speed-ups ranging from $\times 1.50$ to $\times 5.14$. The average speed-up is $\approx \times 3.08$, with larger and more complex circuits (e.g., `2bitAdderMaj`, `parity`, `xor5Maj`) showing the greatest benefit. These results confirm that multithreading significantly accelerates solution discovery, especially for larger search spaces, where creating each partial layout requires more placement steps.

D. *Illustrative Examples*

The following three case studies shed light on how *gold* behaves in practice. The first example in Section IV-D1 demonstrates how a simple switch of the cost objective leads to qualitatively different layouts for the same function. The second example in Section IV-D2 opens the algorithm's *black box* and follows its search trajectory: an initial solution is generated almost instantaneously, after which the engine backtracks through the SSG, revisits earlier branching points, and incrementally discovers ever more compact implementations. In the last case study in Section IV-D3 layouts generated by *ortho* and *gold* for the `parity` function are compared to showcase how the proposed algorithm reduces area.

1) Different Cost Objectives: The first case study shows how the choice of the cost objective steers the engine toward different physical designs.

Example 9. *To illustrate the practical impact of switching cost objectives, we consider a simple 1-bit multiplexer example and create three different layouts with* gold *using the cost functions A (layout area), $|C|$ (number of crossings), and $|W|$ (number of wire segments). The resulting layouts are depicted in Fig. 13 and their metrics are summarized below.*

13a ***Area-optimal:*** $A = 12$, $|C| = 1$, $|W| = 3$. *The layout in Fig. 13a is identical to the solution found by* exact *and therefore minimal in area.*

13b ***Crossing-free:*** $A = 15$, $|C| = 0$, $|W| = 5$. *Eliminating the single crossing of Fig. 13a requires an additional row in Fig. 13b. This solution is* invisible *to* exact*, which is restricted to layouts with minimal area.*

Table V: First legal layout vs. best layout after $100\,\mathrm{s}$ for four effort modes. Metrics: area in tiles (A) and discovery time in seconds (t) with t_{100} showing the time ($\leq 100\,\mathrm{s}$) at which the best-area layout first appeared.

Benchmark [64]		High-Efficiency				High-Effort				Highest-Effort				Maximum-Effort			
Name	$\lvert G\rvert$	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}
c17	18	48	**<0.01**	32	7.88	48	0.01	**28**	31.84	48	0.02	32	4.44	48	0.05	**28**	0.38
t	21	80	**<0.01**	42	15.23	60	0.01	**30**	54.33	60	0.03	42	0.04	36	0.06	32	0.08
t_5	21	63	**<0.01**	**30**	7.71	90	0.01	**30**	10.45	60	0.03	**30**	1.86	42	0.09	**30**	2.50
1bitAdderAOIG	26	90	**<0.01**	70	81.35	90	0.01	72	3.03	90	0.03	72	9.36	90	0.06	**65**	5.47
b1_r2	26	78	**<0.01**	56	21.83	90	0.01	**54**	23.84	90	0.05	**54**	84.71	80	0.09	56	22.83
majority	27	144	**<0.01**	120	68.61	144	0.01	117	71.77	144	0.04	126	0.05	90	0.08	**70**	80.53
majority_5_r1	27	98	**<0.01**	72	96.40	95	0.01	60	2.08	95	0.04	60	8.05	95	0.13	**50**	88.70
newtag	28	80	**<0.01**	60	4.34	80	0.01	60	12.61	80	0.06	60	34.20	80	0.06	**52**	54.93
clpl	30	165	**0.01**	128	0.87	90	**0.01**	90	**0.01**	90	0.07	90	0.07	90	0.12	**70**	21.96
XOR5_R1	40	140	**0.01**	133	0.04	135	0.02	**90**	2.23	135	0.09	**90**	6.70	248	0.16	**90**	0.66
1bitAdderMaj	45	286	**0.01**	231	1.47	216	0.05	**200**	1.74	238	0.12	**200**	4.20	336	0.19	**200**	10.14
cm82a_5	68	276	0.33	276	0.33	260	**0.03**	**234**	0.32	260	0.18	**234**	0.97	260	0.30	**234**	2.58
2bitAdderMaj	82	399	**0.02**	399	**0.02**	630	0.07	399	0.09	630	0.27	399	0.23	630	0.45	**374**	31.48
xor5Maj	102	968	**0.06**	968	**0.06**	968	0.13	924	0.025	968	0.35	**726**	35.59	968	0.66	**726**	40.49
parity	150	846	**0.06**	549	0.18	1917	0.22	549	0.45	1917	0.60	**504**	1.42	1917	1.05	**504**	2.82

Table VI: First legal layout vs. best layout after $100\,\mathrm{s}$ for different *numExpansions* values. Metrics: area in tiles (A) and discovery time in seconds (t) with t_{100} showing the time ($\leq 100\,\mathrm{s}$) at which the best-area layout first appeared.

Benchmark [64]		*numExpansions*=1				*numExpansions*=2				*numExpansions*=4				*numExpansions*=6				*numExpansions*=10			
Name	$\lvert G\rvert$	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}	A_{first}	t_{first}	A_{100}	t_{100}
c17	18	42	0.06	30	1.83	48	0.10	**28**	0.23	48	0.05	**28**	0.38	48	0.13	**28**	0.65	48	0.10	**28**	0.42
t	21	36	0.08	**30**	0.564	42	0.10	**30**	4.63	36	0.06	32	0.08	36	0.09	32	0.06	36	0.12	32	0.12
t_5	21	42	0.08	**30**	0.05	42	0.10	**30**	0.10	42	0.09	**30**	2.50	42	0.11	**30**	14.70	42	0.10	**30**	21.03
1bitAdderAOIG	26	90	0.05	75	0.18	90	0.06	**60**	64.76	90	0.06	65	5.47	90	0.11	65	10.83	90	0.09	65	11.48
b1_r2	26	72	0.11	66	0.22	80	0.13	**42**	20.29	80	0.09	56	22.83	80	0.14	56	39.81	80	0.15	60	0.18
majority	27	90	0.16	66	2.43	90	0.12	**60**	14.46	90	0.08	70	80.53	90	0.10	75	13.21	90	0.14	75	17.77
majority_5_r1	27	102	0.12	60	2.22	102	0.11	**44**	7.31	95	0.13	50	88.70	95	0.11	54	0.28	95	0.16	54	10.34
newtag	28	80	0.13	52	35.70	80	0.18	**52**	0.21	80	0.06	**52**	54.93	80	0.17	60	7.65	80	0.14	60	12.48
clpl	30	168	0.15	77	76.68	90	0.17	**70**	11.32	90	0.12	**70**	21.96	90	0.15	**70**	22.70	121	0.16	**70**	24.01
XOR5_R1	40	220	0.17	**90**	15.38	248	0.16	**90**	0.34	248	0.16	**90**	0.66	248	0.18	**90**	0.90	248	0.21	**90**	1.05
1bitAdderMaj	45	299	0.12	299	0.12	312	0.19	**200**	1.13	336	0.19	**200**	10.14	336	0.23	**200**	21.23	336	0.23	**200**	29.88
cm82a_5	68	261	0.21	**216**	6.24	260	0.28	234	0.97	260	0.30	234	2.58	260	0.39	234	3.31	260	0.39	234	3.55
2bitAdderMaj	82	851	0.63	437	9.51	630	0.43	432	0.53	630	0.45	**374**	31.48	630	0.49	**374**	64.42	630	0.40	378	50.15
xor5Maj	102	*timeout limit reached*				989	2.11	900	30.57	968	0.66	**726**	40.49	968	0.65	**726**	64.84	968	0.70	726	76.66
parity	150	2144	1.21	603	69.16	1917	1.28	531	2.24	1917	1.05	**504**	2.82	1917	1.22	**504**	5.61	1917	1.08	**504**	15.59

Table VII: Impact of multithreading. Metrics: area in tiles (A) and discovery time in seconds (t) with t^{single} showing the time ($\leq 100\,\mathrm{s}$) at which the best-area layout first appeared with single-threading and t^{multi} showing the time it took to find the same solution with multithreading using 10 cores. Speed-up is defined as $\frac{t^{\text{single}}}{t^{\text{multi}}}$.

Benchmark Name [64]	$\lvert G\rvert$	A	t^{single}	t^{multi}	Speed-up
c17	18	28	0.79	0.38	×2.08
t	21	32	0.12	0.08	×1.50
t_5	21	30	4.93	2.50	×1.97
1bitAdderAOIG	45	65	13.09	5.47	×2.39
b1_r2	26	56	74.80	22.83	×3.28
majority	27	75	20.60	5.81	×3.55
majority_5_r1	27	54	0.47	0.20	×2.35
newtag	28	60	14.25	5.62	×2.54
clpl	30	70	74.56	21.96	×3.40
XOR5_R1	40	90	1.75	0.66	×2.65
1bitAdderMaj	45	200	35.93	10.14	×3.54
cm82a_5	68	234	9.24	2.58	×3.58
2bitAdderMaj	82	378	59.01	11.47	×5.14
xor5Maj	102	759	6.82	1.74	×3.92
parity	150	504	13.09	2.82	×4.64

13c ***Wire-minimal:*** *$A = 15$, $|C| = 0$, $|W| = 2$. By relocating one primary input to the fourth row in Fig. 13c, the routing is shortened by two wire segments, achieving the best wiring metric, while preserving zero crossings and the same layout area as in Fig. 13b.*

This example highlights two key properties of *gold*. First, the engine can reproduce or is very close to the area optimum. Second, by relaxing that single objective it explores alternative solutions, discovering layouts that significantly improve crossings or number of wire segments, which cannot be found by any other algorithm. Well-defined cost objectives therefore matter: although the layouts in Fig. 13b and Fig. 13c are both crossing-free, using the objective $|W|$ reveals a superior implementation.

2) Backtracking through SSGs: The second case study shows how the layouts are refined over time by backtracking through the SSGs.

Example 10. *Fig. 14 records the sequence in which* gold(A) *finds valid layouts for the* `newtag` *benchmark. The search first yields a layout with a 16×5 arrangement of tiles, illustrated in Fig. 14a. Whenever the current node in the SSG becomes a dead end or whenever a promising alternative is detected further up the tree, the algorithm backtracks, restores the corresponding partial layout, and explores a different gate placement, as also seen in Fig. 8.*

Each detour uncovers a layout that is strictly smaller than the best one known so far: from 80 tiles (Fig. 14a) gold

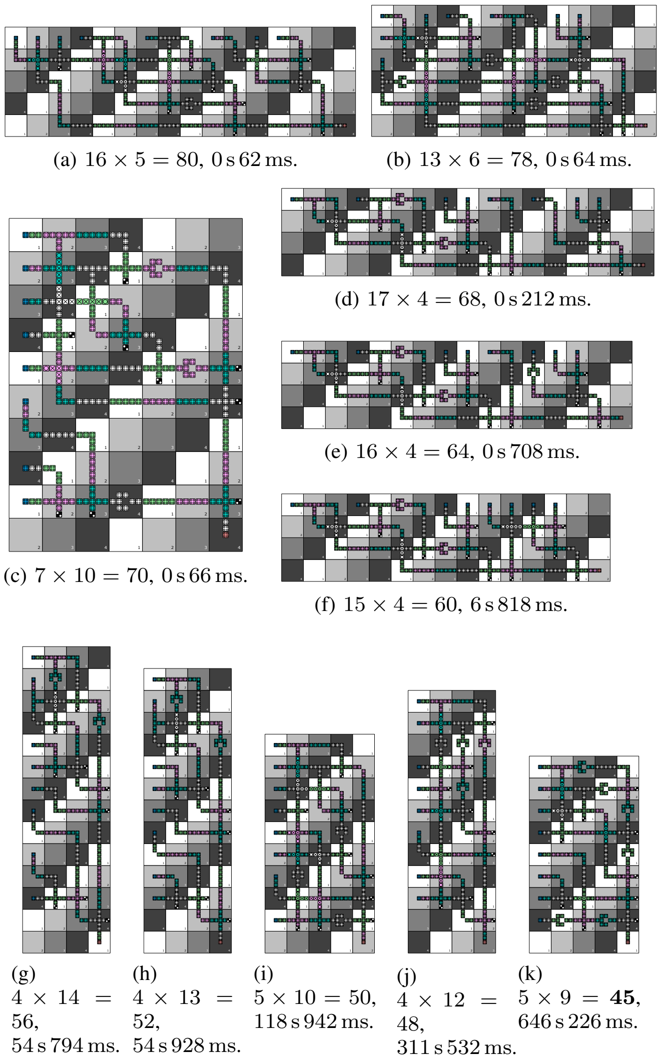

(a) $16 \times 5 = 80$, 0 s 62 ms. (b) $13 \times 6 = 78$, 0 s 64 ms. (c) $7 \times 10 = 70$, 0 s 66 ms. (d) $17 \times 4 = 68$, 0 s 212 ms. (e) $16 \times 4 = 64$, 0 s 708 ms. (f) $15 \times 4 = 60$, 6 s 818 ms. (g) $4 \times 14 = 56$, 54 s 794 ms. (h) $4 \times 13 = 52$, 54 s 928 ms. (i) $5 \times 10 = 50$, 118 s 942 ms. (j) $4 \times 12 = 48$, 311 s 532 ms. (k) $5 \times 9 = \mathbf{45}$, 646 s 226 ms.

Fig. 14: Progressive improvement of the layout for the `newtag` circuit: the first solution (a) is produced in a few milliseconds; subsequent layouts (b–k) are uncovered by backtracking through the SSGs and exploring alternative partial placements, ultimately reducing the area from 80 to **45** tiles.

improves to 78, 70, 68, 64, 60, 56, 52, 50, 48, *and finally* **45** *tiles (Fig. 14k).*

In practice, the first complete layout typically appears within a few milliseconds, providing an early estimate of the layout area. The remaining runtime is spent on guided exploration to determine even smaller layouts.

3) Layout Comparison: Fig. 15 underlines the key difference between the two engines: for the benchmark function `parity`, *ortho* produces a large, sparse layout because each additional gate tends to introduce a new row or column, while *gold* keeps the physical placement close to the netlist structure, packing gates tightly and saving a substantial amount of area.

E. Discussion & Outlook

The experimental results show that *gold* produces near-optimal layouts for small and medium circuits, delivers significant gains in area, crossings, and wiring on larger designs compared to the best heuristic, and completes each run in under a minute, making it a compelling engine for highly optimized, large-scale standard-cell design. Future work will explore a hierarchical flow in which *gold* not only designs individual layouts but also orchestrates the placement of sub-layouts within a top-level network. Additional user-visible parameters, e. g., fixed positions or explicit ordering of primary inputs and outputs, could further ease the embedding of FCN blocks into larger systems whose I/O locations are predetermined.

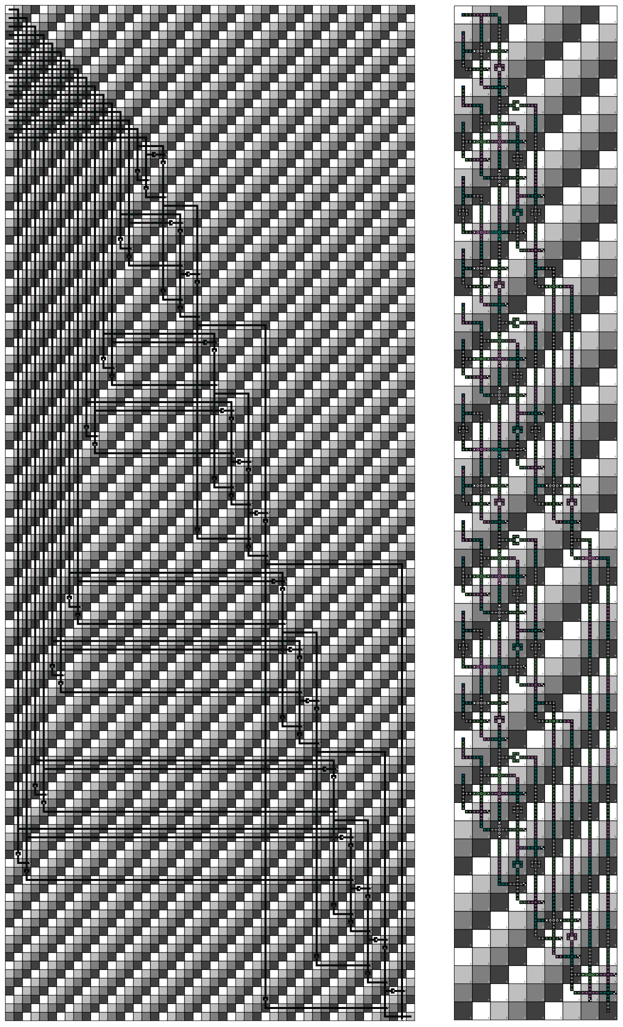

(a) *ortho* (5712 tiles). (b) *gold* (504 tiles).

Fig. 15: Layouts generated for the `parity` circuit. The heuristic *ortho* adds an extra row or column for almost every gate, inflating the footprint, whereas the proposed *gold* engine places gates in a pattern that mirrors the logical network, sharing rows and columns whenever possible and thereby cutting the area by more than an order of magnitude.

V. Conclusion

Recent progress in device fabrication, simulation, and software tools has propelled *Field-coupled Nanocomputing* (FCN) toward practical, post-CMOS applications, but only if the physical design stage can keep pace.

In this paper, we introduced the first FCN layout engine that combines graduated *effort modes* with discretionary cost objectives. Designers can trade runtime for solution quality on demand and steer the search with any weighted cost objective, thereby embedding technology-specific knowledge, such as simulation data or fabrication limits, directly into the layout generation step.

An open-source implementation, integrated into the *Munich Nanotech Toolkit* (MNT), scales to circuits beyond the reach of state-of-the-art exact solvers. On those benchmarks it reduces layout area by an average of 73.07 %, number of crossings by 19.10 %, and number of wire segments by 54.47 % relative to a state-of-the-art heuristic baseline. Even after applying post-layout optimization to the layouts generated by the heuristic, our approach still achieves average gains of 25.99 % in area, 37.82 % in crossings, and 25.96 % in wire segments.

These results establish physical design via parallelized multi-objective search space exploration as a cornerstone for future FCN physical design flows and open the door to versatile layout generation using highly optimized, large-scale standard-cell design.

References

[1] N. G. Anderson and S. Bhanja, Eds., *Field-Coupled Nanocomputing - Paradigms, Progress, and Perspectives*. Springer, 2014.

[2] J. Pitters *et al.*, "Atomically Precise Manufacturing of Silicon Electronics," *ACS Nano*, 2024.

[3] R. Achal *et al.*, "Lithography for robust and editable atomic-scale silicon devices and memories," *Nat. Commun.*, vol. 9, no. 1, 2018.

[4] J. Drewniok *et al.*, "*QuickSim*: Efficient *and* Accurate Physical Simulation of Silicon Dangling Bond Logic," in *IEEE-NANO*, 2023.

[5] S. S. H. Ng *et al.*, "SiQAD: A Design and Simulation Tool for Atomic Silicon Quantum Dot Circuits," *IEEE TNANO*, vol. 19, pp. 137–146, 2020.

[6] J. Drewniok *et al.*, "Minimal Design of SiDB Gates: An Optimal Basis for Circuits Based on Silicon Dangling Bonds," in *NANOARCH*, 2023.

[7] R. Lupoiu *et al.*, "Automated Atomic Silicon Quantum Dot Circuit Design via Deep Reinforcement Learning," *ArXiv*, vol. abs/2204.06288, 2022.

[8] J. Drewniok *et al.*, "The Need for Speed: Efficient Exact Simulation of Silicon Dangling Bond Logic," in *ASP-DAC*, 2024, pp. 576–581.

[9] Y. Ardesi *et al.*, "SCERPA: A Self-Consistent Algorithm for the Evaluation of the Information Propagation in Molecular Field-Coupled Nanocomputing," *TCAD*, vol. 39, no. 10, pp. 2749–2760, 2020.

[10] M. Walter *et al.*, "Reducing the Complexity of Operational Domain Computation in Silicon Dangling Bond Logic," in *NANOARCH*, 2023.

[11] J. Drewniok *et al.*, "Temperature Behavior of Silicon Dangling Bond Logic," in *IEEE NANO*, 2023, pp. 925–930.

[12] M. Walter *et al.*, "An Exact Method for Design Exploration of Quantum-dot Cellular Automata," in *DATE*, 2018, pp. 503–508.

[13] S. Hofmann *et al.*, "A* is Born: Efficient and Scalable Physical Design for Field-coupled Nanocomputing," in *IEEE-NANO*, 2024, pp. 80–85.

[14] M. Walter *et al.*, "Scalable Design for Field-Coupled Nanocomputing Circuits," in *ASP-DAC*, 2019, pp. 197–202.

[15] S. Hofmann *et al.*, "Post-Layout Optimization for Field-coupled Nanotechnologies," in *NANOARCH*, 2023.

[16] M. Walter *et al.*, "Versatile Signal Distribution Networks for Scalable Placement and Routing of Field-coupled Nanocomputing Technologies," in *ISVLSI*, 2023.

[17] S. Hofmann *et al.*, "Late Breaking Results: Wiring Reduction for Field-coupled Nanotechnologies," in *DAC*, 2024.

[18] Y. Li *et al.*, "Field-Coupled Nanocomputing Placement and Routing with Genetic and A* Algorithms," *IEEE TCAS-I*, vol. 69, no. 11, pp. 4619 – 4631, 2022.

[19] S. Hofmann *et al.*, "Scalable Physical Design for Silicon Dangling Bond Logic: How a 45° Turn Prevents the Reinvention of the Wheel," in *IEEE-NANO*, 2023, pp. 872–877.

[20] G. Li *et al.*, "A QCA Placement and Routing Algorithm Based on the SA Algorithm," *Int. J. Electron*, 2023.

[21] S. Hofmann *et al.*, "Late Breaking Results From Hybrid Design Automation for Field-coupled Nanotechnologies," in *DAC*, 2023.

[22] B. Zhang *et al.*, "Quantum-dot Cellular Automata Placement and Routing with Hierarchical Algorithm," *Nano Commun. Netw.*, vol. 39, p. 100495, 2024.

[23] S. Hofmann *et al.*, "Physical Design for Field-coupled Nanocomputing with Discretionary Cost Objectives," in *LASCAS*, 2025.

[24] F. Peng *et al.*, "Spars: A Full Flow Quantum-Dot Cellular Automata Circuit Design Tool," *TCAS-II*, vol. 68, no. 4, pp. 1233–1237, 2021.

[25] S. Hofmann *et al.*, "Thinking Outside the Clock: Physical Design for Field-coupled Nanocomputing with Deep Reinforcement Learning," in *ISQED*, 2024.

[26] R. E. Formigoni *et al.*, "A Survey on Placement and Routing for Field-Coupled Nanocomputing," *JICS*, vol. 16, pp. 1–9, 2021.

[27] M. Walter and R. Wille, "Efficient Multi-Path Signal Routing for Field-coupled Nanotechnologies," in *NANOARCH*, 2022.

[28] S. Hofmann *et al.*, "Efficient and Scalable Post-Layout Optimization for Field-coupled Nanotechnologies," *TCAD*, 2025.

[29] M. Walter *et al.*, "One-pass Synthesis for Field-coupled Nanocomputing Technologies," in *ASP-DAC*, 2021, pp. 574–580.

[30] K. Walus *et al.*, "QCADesigner: A Rapid Design and Simulation Tool for Quantum-Dot Cellular Automata," *TNANO*, vol. 3, no. 1, pp. 26–31, 2004.

[31] F. Riente *et al.*, "MagCAD: Tool for the Design of 3-D Magnetic Circuits," *JXCDC*, vol. 3, pp. 65–73, 2017.

[32] R. E. Formigoni *et al.*, "Ropper: A Placement and Routing Framework for Field-Coupled Nanotechnologies," in *SBCCI*. ACM, 2019.

[33] F. Riente *et al.*, "ToPoliNano: A CAD Tool for Nano Magnetic Logic," *TCAD*, vol. 36, no. 7, pp. 1061–1074, 2017.

[34] M. Walter *et al.*, "fiction: An Open Source Framework for the Design of Field-coupled Nanocomputing Circuits," 2019, arXiv:1905.02477.

[35] S. Hofmann *et al.*, "Late Breaking Results: Physical Co-Design for Field-coupled Nanocomputing," in *DATE*, 2025.

[36] Y. Li *et al.*, "iFCN: Automated Design Platform for Molecular FCN Circuits," https://github.com/li-yangshuai/iFCN, 2025.

[37] J. Drewniok *et al.*, "Unifying Figures of Merit: A Versatile Cost Function for Silicon Dangling Bond Logic," in *IEEE-NANO*, 2024, pp. 91–96.

[38] F. S. Torres *et al.*, "Evaluating the Impact of Interconnections in Quantum-Dot Cellular Automata," in *DSD*, 2018, pp. 649–656.

[39] A. Chaudhary *et al.*, "Fabricatable Interconnect and Molecular QCA Circuits," *TCAD*, vol. 26, no. 11, pp. 1978–1991, 2007.

[40] F. Kashfi *et al.*, "Multi-Objective Optimization Techniques for VLSI Circuits," in *ISQED*, 2011.

[41] R. Martins *et al.*, "Multi-objective optimization of analog integrated circuit placement hierarchy in absolute coordinates," *Expert Systems with Applications*, vol. 42, no. 23, pp. 9137–9151, 2015.

[42] M. Walter *et al.*, "The Munich Nanotech Toolkit (MNT)," in *IEEE-NANO*, 2024.

[43] S. Hofmann *et al.*, "MNT Bench: Benchmarking Software and Layout Libraries for Field-coupled Nanocomputing," in *DATE*, 2024.

[44] C. Lent *et al.*, "Quantum Cellular Automata: The Physics of Computing with Arrays of Quantum Dot Molecules," in *PhysComp*, 1994, pp. 5–13.

[45] K. Hennessy and C. S. Lent, "Clocking of Molecular Quantum-dot Cellular Automata," *J. Vac. Sci. Technol. B*, vol. 19, no. 5, pp. 1752–1755, 2001.

[46] C. Lent and P. Tougaw, "A Device Architecture for Computing with Quantum Dots," *Proc. IEEE*, vol. 85, no. 4, pp. 541–557, 1997.

[47] D. A. Reis *et al.*, "A Methodology for Standard Cell Design for QCA," in *ISCAS*, 2016, pp. 2114–2117.

[48] T. Huff *et al.*, "Binary atomic silicon logic," *Nat. Electron.*, vol. 1, no. 12, pp. 636–643, 2018.

[49] M. B. Haider *et al.*, "Controlled Coupling and Occupation of Silicon Atomic Quantum Dots at Room Temperature," *Phys. Rev. Lett.*, vol. 102, p. 046805, 2009.

[50] T. Huff *et al.*, "Atomic White-Out: Enabling Atomic Circuitry through Mechanically Induced Bonding of Single Hydrogen Atoms to a Silicon Surface," *ACS Nano*, vol. 11 9, pp. 8636–8642, 2017.

[51] R. A. Wolkow *et al.*, "Silicon Atomic Quantum Dots Enable Beyond-CMOS Electronics," in *Field-Coupled Nanocomputing*, 2013.

[52] N. Pavliček *et al.*, "Tip-induced passivation of dangling bonds on hydrogenated Si(100)-2$\times$1," *APL*, vol. 111, no. 5, p. 053104, 2017.

[53] M. Rashidi *et al.*, "Initiating and Monitoring the Evolution of Single Electrons Within Atom-Defined Structures," *PRL*, vol. 121, p. 166801, 2018.

[54] M. Walter *et al.*, "Hexagons are the Bestagons: Design Automation for Silicon Dangling Bond Logic," in *DAC*, 2022, pp. 739–744.

[55] B. Hien *et al.*, “Reducing Wire Crossings in Field-Coupled Nanotechnologies,” in *IEEE-NANO*, 2024, pp. 155–160.
[56] F. Sill Torres *et al.*, “On the Impact of the Synchronization Constraint and Interconnections in Quantum-dot Cellular Automata,” *MICPRO*, vol. 76, pp. 103–109, 2020.
[57] V. Vankamamidi *et al.*, “Clocking and Cell Placement for QCA,” in *IEEE-NANO*, vol. 1, 2006, pp. 343–346.
[58] C. Campos *et al.*, “USE: A Universal, Scalable and Efficient clocking scheme for QCA,” *IEEE TCAD*, vol. 35, pp. 513–517, 2016.
[59] M. Goswami *et al.*, “An Efficient Clocking Scheme for Quantum-dot Cellular Automata,” *Int. J. Electron. Lett.*, vol. 8, no. 1, pp. 83–96, 2020.
[60] M. Walter *et al.*, “Placement and Routing for Tile-Based Field-Coupled Nanocomputing Circuits Is NP-Complete (Research Note),” *JETC*, vol. 15, no. 3, 2019.
[61] P. E. Hart *et al.*, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
[62] B. Steinbach, *Recent Progress in the Boolean Domain*. Cambridge Scholars Publishing, 2014.
[63] A. Trindade *et al.*, “A Placement and Routing Algorithm for Quantum-dot Cellular Automata,” in *SBCCI*, 2016, pp. 1–6.
[64] G. Fontes *et al.*, “Placement and Routing by Overlapping and Merging QCA Gates,” in *ISCAS*, 2018.
[65] K. McElvain, “IWLS’93 Benchmark Set: Version 4.0,” 1993.
[66] M. Walter *et al.*, “Verification for Field-coupled Nanocomputing Circuits,” in *DAC*, 2020.

Simon Hofmann (Graduate Student Member, IEEE) received the M.S. degree in electrical engineering from the Technical University of Munich (TUM), Munich, Germany, in 2022. He is currently pursuing the Ph.D. degree with the Chair for Design Automation, TUM, and is also a Quantum Software Engineer with the Munich Quantum Software Company (MQSC), Garching near Munich, Germany. His research interests include design automation for Field-coupled Nanotechnologies (FCN).

Marcel Walter (Member, IEEE) received the Ph.D. degree in computer science from the University of Bremen, Bremen, Germany, in 2021 for his work on algorithms for the physical design of emerging post-CMOS nanotechnologies. He is currently a Postdoctoral Researcher with the Technical University of Munich and a Senior Quantum Software Engineer with the Munich Quantum Software Company (MQSC). In 2024, he served as a substitute professor at the University of Bremen. Furthermore, he is the initiator and maintainer of the "fiction" framework for the logic synthesis, physical design, verification, and simulation of Field-coupled Nanotechnologies, as well as the "aigverse" library that bridges the gap between machine learning and logic synthesis.

Robert Wille (Senior Member, IEEE) is a Full and Distinguished Professor with the Technical University of Munich, CEO of the Munich Quantum Software Company (MQSC), and Scientific Director with the Software Competence Center Hagenberg. His research focuses on the design of circuits and systems for both conventional and emerging technologies, with over 15 years of contributions to quantum computing—particularly in foundational software and design automation. He has received numerous accolades, including Best Paper Awards, the DAC Under-40 Innovator Award, a Google Research Award, and an ERC Consolidator Grant. He collaborates with leading academic and industrial partners, plays a key role in initiatives such as the Munich Quantum Valley, and actively supports technology transfer through his roles in industry. He has published over 400 papers and serves on editorial and advisory boards of major journals and conferences.