# Fine-tuning CLB Placement to Speed up Reconfigurations in NVM-based FPGAs

Yuan Xue[1] Patrick Cronin[1] Chengmo Yang[1] Jingtong Hu[2]

[1]Department of Electrical and Computer Eng.
University of Delaware
Newark, DE 19716 USA
{xueyuan,ptrick,chengmo}@udel.edu

[2]School of Electrical and Computer Eng.
Oklahoma State University
Stillwater, OK 74078 USA
jthu@okstate.edu

***Abstract*—Non-volatile memories (NVMs) outperform traditional SRAMs in terms of low power consumption, high capacity, near-zero power-on delay, and high error-resistance. Researchers have demonstrated the possibilities of implementing FPGA building blocks with various types of NVMs. However, NVMs also bring several new design challenges to FPGAs: the slow write performance of NVM may degrade FPGA (re)configuration speed, while the limited write endurance of NVM constrains the number of times that the FPGA can be (re)configured. Unfortunately, none of these NVM features are taken into consideration in current FPGA synthesis tools, which have been optimized solely for SRAM-based FPGAs. To tackle this limitation, we propose to make the FPGA placement process aware of the slow and costly NVM writes. Our contributions are three-fold: We first construct mathematical models to characterize reconfiguration costs in NVM-based FPGAs. Second, we identify three types of flexibilities that can be exploited to reduce the reconfiguration cost. Finally, we present three approaches for designers to fine-tune the placement process to balance the reconfiguration cost and traditional timing and routability constraints according to their needs. The proposed algorithms are incorporated in Verilog-to-Routing (VTR) CAD tool. Experiments on standard MCNC benchmark circuits show that our approach eliminates up to 67% NVM writes during the reconfiguration process, thus effectively improving the performance and endurance of NVM-based FPGAs.**

***Keywords*—*self-adaptive system, reconfiguration overhead, non-volatile memory-based FPGA, CLB placement.***

## I. Introduction

Self-adaptivity is a key requirement for many embedded and cyber-physical systems to consistently interact with dynamic, uncertain, and noisy physical environments. Field Programmable Gate Arrays (FPGAs), being reconfigurable, are a natural platform to deliver a high-speed implementation of these applications. In current FPGA design, both on-chip reconfigurable components and storage blocks are implemented with SRAM. Configuration information is stored in off-chip Flash memory, and loaded to on-chip SRAM when the FPGA is powered on. While the fast write/read speed of SRAM accelerates FPGA reconfiguration and performance, it unfortunately also leads to four fundamental drawbacks. First, the large size of on-chip SRAM makes the FPGA power-hungry, since technology scaling causes leakage power to increase dramatically and dominate total power dissipation [1]. Second, the limited scalability of SRAM capacity constrains the number of on-chip reconfigurable components and storage blocks, which in turn constrains the size of FPGA-implementable designs. Third, SRAM is vulnerable to radiation-induced random bit flips, leading to not only transient data errors, but also "firm" logic errors that change the functionality of the design [2], [3], [4]. Finally, since SRAM is volatile, upon a power outage, all the on-chip configuration information becomes unavailable and needs to be reloaded from off-chip storage when the power is back on, which is harmful to FPGA performance and configuration time.

Recently researchers started to exploit the use of Non-Volatile Memory (NVM) in FPGAs [1], [5], [6], [7], [8]. Compared to SRAM, NVMs have much higher capacity, superior energy efficiency, high error-resistance and near-zero power-on delay. All of these characteristics are attractive to FPGAs: more logic and storage cells can be implemented on a single FPGA chip, which consumes much less power and is more resilient to power interruptions. However, the characteristics of NVMs also pose two new challenges for their usage in self-adaptive systems implemented with FPGAs. First, many self-adaptive systems need to reconfigure or evolve quickly. Unfortunately, the slow write performance of NVMs makes reconfiguration time non-trivial. Second, due to the limited write endurance of NVMs, frequently reconfiguring the FPGA may wear some building blocks heavily and cause stuck-at faults [9]. For instance, Actel's Flash-based FPGA only provides 500 programming cycles [10], which clearly are not able to fulfill the needs of self-adaptive systems.

Existing FPGA synthesis and reconfiguration techniques are still developed for SRAM-based FPGAs. Applying them directly to NVM-based FPGAs will yield inferior results. To overcome this limitation, we propose to revise FPGA synthesis flow according to the distinctive features of NVMs. Specifically, we propose to fine-tune the placement process to minimize the (re)configuration cost, which is determined by the total number of NVM cells programmed during (re)configuration. Before programming a block, the original content is read out and compared with the new content, and the identical part is eliminated from programming. To maximize this type of content reuse, we exploit the flexibilities inherent in placing Look-up Tables (LUTs). In a nutshell, the main contributions of this paper include:

- Construction of a mathematical model for characterizing the cell-level reconfiguration cost of NVM-based LUTs.
- Identification of three types of flexibilities that can be exploited to reduce the reconfiguration cost.
- Development of three approaches for designers to fine-tune the placement process to balance the reconfiguration cost and traditional timing and routability constraints according to their needs.
- Incorporation of the proposed approaches into the Verilog-to-Routing (VTR) CAD tool [11] to demonstrate their feasibility and effectiveness.

To the best of our knowledge, this is the first work that studies reconfiguration optimizations in NVM-based FPGAs. In the rest of this paper, we will first present the background knowledge of NVM-based FPGAs and related works in Section II, and then outline the technical motivation and challenges in Section III. Section IV details the proposed reconfiguration cost model and the three types of flexibilities. Section V discusses the three NVM-aware placement approaches. Section VI presents the experimental results, while Section VII concludes the paper.

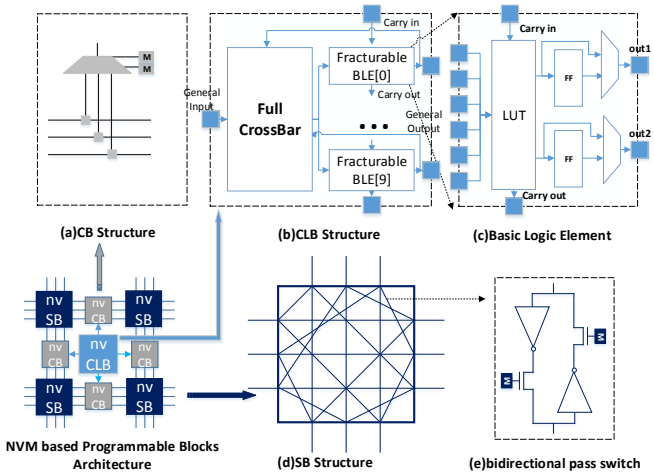


Figure 1. FPGA block structures, including (a) connection blocks, (b) CLB, (c) BLE, (d) Wilton switch box [17] and (e) bidirectional pass transistor switch. LUTs and cells denoted with 'M' can be implemented with NVMs.

## II. Background

### A. NVM-based FPGAs

Non-volatile Memories (NVMs), including Flash memory, Phase Change Memory (PCM), Spin Transfer Torque Magnetic RAM (STT-MRAM) and Ferroelectric RAM (FRAM), use physical characteristics (typically resistance or magnetic field) to represent logic states. NVMs offer many advantages such as high density, near-zero power-on delay, high error-resistance, and superior energy efficiency [1], all of which can benefit FPGAs. Researchers have proposed to use NVMs instead of SRAM as FPGA on-chip storage, such as Flash-based FPGAs [5], PCM-based FPGAs [1], [8], and MRAM-based FPGAs [7]. They have demonstrated the possibilities of using NVM cells to implement different FPGA building blocks, including configurable logic blocks (CLBs), switch boxes (SBs), and connection blocks (CBs). Figure 1 gives a representative FPGA architecture with structure details of CLB, SB, and CB. A CLB (Figure 1(b)) is composed of 10 Basic Logic Elements (BLEs) connected with a full crossbar, which implements arbitrary connections across the BLEs. Each BLE (Figure 1(c)) contains one LUT, two flip-flops, and two multiplexers. LUTs in Figure 1(c) and cells denoted with 'M' in Figure 1(a)(e) contain configuration information and can be implemented with NVM cells.

### B. Reconfiguration Optimizations

Many commercial FPGA products support *partial reconfiguration* (PR) wherein hardware resources can be multiplexed across different tasks [12] at runtime. Utilizing such capacity, some techniques tried to reusing FPGA blocks through design partitioning and/or scheduling [13], [14]. Based on coarse-grained functional information, these techniques partitioned a design into static and reconfigurable parts, aiming at minimizing sizes of the reconfigurable parts.

Another set of techniques tried to maximize similarity between two designs and reuse logic at the register transfer level. A functional-level comparison approach was proposed in [15], aiming at reusing entire logic components such as ALUs and float-point units. Mehdipour *et al.* [16] proposed an algorithm that divides a design into several tasks, aiming at finding one or a pair of identical operations across these tasks. A dummy node insertion algorithm was further used to increase the number of similar pairs.

Logic reuse has also been exploited at the LUT level. These techniques characterize similarity between two designs with graph matching or topological structure analysis. A logic remapper was proposed in [18]. It first measured topological similarity by the number of matching connections between two matching logic blocks, and then performed linear programming to minimize the routing cost. In [19], similarity between two nodes is defined as the weighted sum of the position similarities between them and their adjacent nodes. Since the target platform is an SRAM-based FPGA, these techniques abstract LUTs as nodes without considering their contents. In other words, a LUT is "reusable" only if it does not need to be reconfigured at all. Logic similarity check has also been applied at the engineering change order (ECO) stage [20], [21]. The goal of ECO is to detect changes between two largely identical designs. As a result, these techniques mainly focus on checking functional similarity, which is different from the LUT content similarity considered in this paper. Finally, some previous work [22] has considered LUT content not for reconfiguration optimization but for identifying IP cores.

To summarize, all of previous reconfiguration optimizations target SRAM-based FPGAs. They model reconfiguration cost as the number of LUTs and switch boxes to be programmed when implementing a new design/task. In contrast, the proposed technique targets NVM-based FPGAs, and models reconfiguration cost as the number of NVM cells to be programmed, which is at a much finer granularity than existing approaches. In the next section, we will explain the underlying reasons that motivate us towards this cell-level reconfiguration optimization framework.

Table I. Comparison of SRAM and various NVM cells [9]

| | Cell elements | Area ($nm^2$) | Read time (ns) | Write time (ns) | Write cycles |
|---|---|---|---|---|---|
| SRAM | 6T | 283,500 | 0.2 | 0.2 | $10^{16}$ |
| PCM | 1T1R | 8,100 | 12 | 100 | $10^{9}$ |
| STT-MRAM | 1T1R | 84,500 | 35 | 35 | $10^{12}$ |
| NOR Flash | 1T | 20,250 | 15 | 1000 | $10^{5}$ |

## III. Technical Motivation

This section presents the necessity for minimizing the reconfiguration cost in NVM-based FPGAs as well as the challenges and opportunities.

Feasibility of replacing the on-chip SRAM in FPGAs with NVMs has been demonstrated in [1], [6], and [8]. However, the long write latency of NVMs makes the reconfiguration time non-trivial. Table I shows the comparison between SRAM and various NVM cells, including PCM, STT-MRAM, and NOR Flash. Among these NVM technologies, PCM displays the highest density, the fastest read speed, and relatively high write cycles. However, its write speed is 500 times slower than SRAM. This will become the bottleneck when PCM replaces SRAM as on-chip storage, especially when the FPGA needs to be frequently reconfigured, as required by many self-adaptive applications.

To qualify PCM-based FPGAs for self-adaptive systems, it is necessary to minimize the (re)configuration cost, which is determined by the number of PCM cells to program during (re)configuration. Among many techniques that mitigate the adverse impact of PCM write operations [23], [24], [25], one that can be directly applied to NVM-based FPGAs is *redundant write elimination* [26]: before programming a LUT, the original content can be read out, and the identical part in the new content can be excluded from programming. This technique was not adopted in SRAM-based FPGAs since SRAM read and write operations have similar speed and overhead. However, for PCM, Table I shows that write operations are more than 8 times slower and more costly than read operations. As a result, the *redundant write elimination* strategy is effective for reducing the

reconfiguration cost, with the amount of reduction proportional to the number of NVM write operations eliminated.

Overall, the consideration of the reconfiguration cost in NVM-based FPGAs motivates the proposal of a framework that considers two designs (or two tasks in one design), one currently on the FPGA and one to be configured, and minimizes the bit-flips when configuring the new design. The framework should address three questions. Given two designs, how to model the reconfiguration cost? What types of flexibilities can be exploited to reduce this cost? How to ensure that traditional area, routing, and timing constraints can still be fulfilled? In the rest of this paper, these questions will be addressed one by one.

## IV. Reconfiguration Cost Models and Flexibilities

This section first presents the proposed (re)configuration cost model for NVM-based LUTs, and then identifies three types of flexibilities that can be exploited to reduce the (re)configuration cost.

### A. Reconfiguration Cost Model

One significant difference between SRAM-based and NVM-based FPGAs is the granularity at which the reconfiguration cost can be characterized. In SRAM-based FPGAs, the reconfiguration cost is proportional to the number of *blocks* to program when implementing a design/task. However, in NVM-based FPGAs, when programming a block, the identical part in the content can be excluded from being overwritten. As a result, the reconfiguration cost should be modeled at a finer-granularity, specifically, as the number of *NVM cells* to write when implementing a new design.

At first sight, it seems that the reconfiguration cost can be easily obtained through a bitwise comparison between the bitstreams of two designs. However, the proposed framework does not adopt this approach due to two reasons. First, in commercialized FPGAs, bitstreams are in encrypted form and hence cannot be directly compared to obtain the number of bit-flips required to configure the new design. Second, even if decrypted bitstreams are available, comparing them only gives the total reconfiguration cost without any detailed information of where the bottleneck is. Therefore, such information is not useful for guiding FPGA synthesis procedures for reducing the reconfiguration cost.

Because of these reasons, we propose a much finer-grained model that characterizes the cost of configuring *configurable logic blocks* (CLBs) in an NVM-based FPGA. As shown in Figure 1(b)(c), CLBs are composed of LUTs that can be implemented with NVM cells [26]. Therefore the proposed framework models CLB reconfiguration cost as the sum of the number of bit-flips required to program each LUT in it. Specifically, a $k$-input LUT has $2^k$ entries, thus requiring $2^k$ NVM cells for storing its configuration[1]. Given two LUTs, if $p$ specified bits (i.e., not *don't care* bits) are different, the cost of configuring one to the other is $p$. The proposed framework therefore uses the following set of equations to model the reconfiguration cost of a LUT (denoted as $RR_{LUT}$), a CLB (denoted as $RR_{CLB}$), and the total reconfiguration cost of CLBs (denoted as $Cost_{\text{reconfig}}$). Here, $N_{CLB}$ denotes the total number of CLBs in the new design.

$$RR_{LUT} = p \tag{1}$$

$$RR_{CLB} = \sum_{LUT_i \in CLB} RR_{LUT_i} \tag{2}$$

$$Cost_{\text{reconfig}} = \sum_{i=1}^{N_{CLB}} RR_{CLB_i} \tag{3}$$

[1]We assume single-level-cell (SLC) NVMs in this paper. However the model can be easily extended for multi-level-cell (MLC) NVMs [27], [28] that store more than one bit per cell.

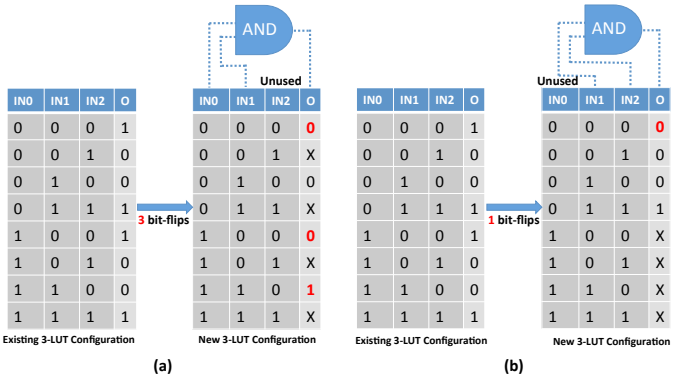


Figure 2. Intra-LUT reconfiguration flexibility. In (a), the AND gate uses IN0 and IN1 as inputs and reconfiguration cost is 3 bit-flips. In (b), the AND gate uses IN1 and IN2, and reconfiguration cost is reduced to 1 bit-flip.

Figure 2(a) shows a concrete example of the LUT reconfiguration cost. The left LUT belongs to the design currently on the FPGA and hence is fully specified, while the right LUT belongs to the design to be configured and hence includes *don't care* bits (denoted as 'X'). In this example, three bit-flips are required to implement the right LUT (i.e., $p = 3$).

Since the reconfiguration cost depends on the positions of LUTs, it will be influenced by the placement procedure. By manipulating the flexibilities inherent during placement, it is possible to reduce the reconfiguration cost. In the next subsection, we will identify several types of flexibilities for reducing the CLB reconfiguration cost, which dominates the total reconfiguration cost according to our experiments.

### B. Reconfiguration Flexibilities

Reconfiguration flexibility refers to the different possibilities of implementing a design on the NVM-based FPGA. In terms of placement, there exist many ways both in mapping logical blocks to physical blocks and in configuring a physical block to implement a given logical block. In a nutshell, three types of flexibilities can be exploited to reduce the reconfiguration cost:

- At *intra-LUT* level, the flexibility of manipulating orders and positions of LUT inputs is helpful for reducing $RR_{LUT}$.
- At *inter-LUT* level, the flexibility of switching LUT positions within a CLB is helpful for reducing $RR_{CLB}$.
- At *inter-CLB* level, the flexibility of remapping CLBs within a given area is helpful for reducing $Cost_{\text{reconfig}}$.

Intra-LUT flexibility reflects the logic equivalence of different inputs in a LUT. A concrete example is given in Figure 2. It shows two ways to reconfigure a 3-input LUT whose current content is "10011001" to implement a 2-input AND gate. In Figure 2(a), IN0 and IN1 are used as inputs and IN2 is set to 0. The new content of the LUT is "0x0x0x1x", which requires 3 bit-flips to configure. In contrast, Figure 2(b) shows that by using IN1 and IN2 as inputs and setting IN0 to 0, the reconfiguration cost is reduced to 1 bit-flip. More generally, assume a $k$-input LUT is used to implement an $l$-input ($l \leq k$) gate. There exist $k!/(k-l)!$ possible ways to map the $l$ signals to the $k$ pins. For the unused $k-l$ pins, there exist $2^{k-l}$ possible ways to connect them to 1 or 0. As a result, the total amount of intra-LUT flexibility is:

$$Flex_{\text{intra-LUT}} = 2^{k-l} \times \frac{k!}{(k-l)!} \tag{4}$$

Inter-LUT flexibility exists because each CLB contains a fully connected crossbar capable of implementing arbitrary connections

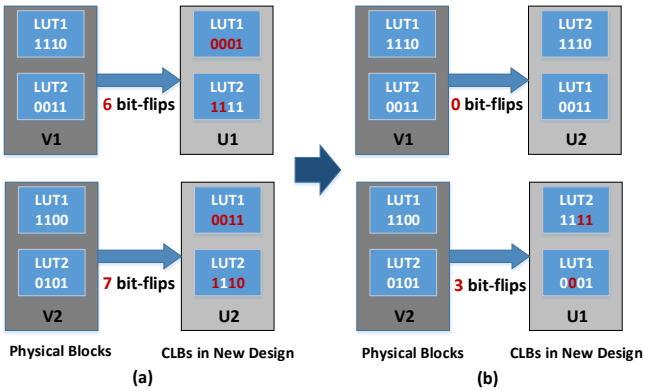


Figure 3. Inter-LUT and Inter-CLB reconfiguration flexibilities. (a) shows the original cost (13 bit-flips) for configuring $U1$ and $U2$. (b) shows by switching $U1$ and $U2$ and $LUT1$ and $LUT2$ in $U2$, the cost is reduced to 3 bit-flips.

across BLEs, as shown in Figure 1(b). As a result, two LUTs in a CLB can be flexibly switched to reduce the reconfiguration cost. Similarly, inter-CLB flexibility reflects the freedom in mapping logical blocks to physical blocks, which is always exploited during placement. Figure 3 shows an example illustrating these two types of flexibilities. The example includes four CLBs, with $V1$ and $V2$ denoting current contents of the physical blocks and $U1$ and $U2$ denoting contents of the logical blocks. Each CLB has two LUTs. Figure 3(a) shows that by directly mapping $U1$ to $V1$ and $U2$ to $V2$, the total reconfiguration cost is 13 bit-flips. On the other hand, the two types of flexibilities imply that both $U1$ or $U2$ can be mapped to either $V1$ and $V2$, and the LUT positions in $U1$ and $U2$ can be switched as well. Figure 3(b) shows that by mapping $U1$ to $V2$ and $U2$ to $V1$ and switching LUT1 and LUT2 in $U2$, the reconfiguration cost is reduced to 3 bit-flips.

Overall, to map $u$ logical blocks to $v$ ($u \leq v$) physical blocks, there exist $v!/(v-u)!$ possibilities. Mapping $m$ logical LUTs to a CLB containing $n$ physical LUTs can be modeled in the same way. Therefore, the total amount of inter-LUT and inter-CLB flexibilities are respectively shown in the following two equations:

$$Flex_{\text{inter-LUT}} = \frac{n!}{(n-m)!} \quad (5)$$

$$Flex_{\text{inter-CLB}} = \frac{v!}{(v-u)!} \quad (6)$$

Since the three types of reconfiguration flexibilities lie at different levels, they can be exploited independently. The overall amount of reconfiguration flexibility is a product of them. While this delivers a large potential for reducing the reconfiguration cost, it also imposes a challenge in efficiently exploiting this huge design space. In the next section, we will show three different approaches for incorporating these flexibilities into the placement procedure of VTR [11].

## V. Reconfiguration-aware Placement

Challenges of reconfiguration-aware placement lie in three aspects: (1) efficiently exploiting the huge design space for reducing the reconfiguration cost, (2) balancing the reconfiguration cost and other design constraints, and (3) providing designers with flexibilities to tune the placement procedure according to their needs. To address these issues, the VTR synthesis flow is revised to take reconfiguration cost into consideration, as shown in Figure 4. First, post-placement information of the design currently implemented on the FPGA is extracted for computing the reconfiguration cost. Second, the placement procedure is replaced with three different approaches, allowing designers to select the more desired one according to their needs:

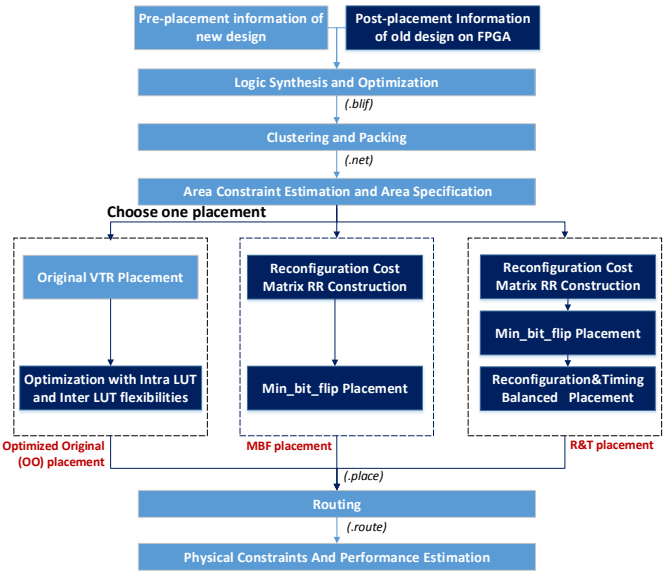


Figure 4. Augmented VTR synthesis flow with three different placement strategies and multiple new steps (dark blue) added to original VTR procedures (light blue).

- The *optimized original* (OO) placement prefers timing and routability over reconfiguration cost. The original timing-driven VTR placement algorithm is adopted to determine CLB positions. Then, intra-LUT and inter-LUT optimizations are applied to each CLB.
- The *min-bit-flip* (MBF) placement prefers reconfiguration cost over timing and routability and exploits all the three types of flexibilities identified in Section IV-B. The original VTR placement is replaced with a bipartite graph model-based approach that generates a placement with minimum CLB reconfiguration cost.
- The *Reconfiguration & Timing balanced* (R&T) placement uses the MBF placement as the initial placement and applies VTR optimizations on it. The cost function of VTR placement is tuned to balance timing and reconfiguration cost using a threshold given by the designer.

One noteworthy property is that all the three placement approaches fulfill the same area constraints. They only differ in their orders for prioritizing reconfiguration cost, timing and routability. The following parts of this section present these approaches one by one.

### A. Optimized Original (OO) Placement

The OO placement adopts the original timing-driven VTR placement algorithm to determine CLB positions. As a result, it does not influence critical path, but is unable to exploit inter-CLB flexibility. In this approach, both the original content and the new content of a CLB is known. For each original-new CLB pair, the challenge is to fully exploit intra-LUT and inter-LUT flexibilities.

A quick examination indicates that inter-LUT flexibility is more complex to exploit than intra-LUT flexibility. The amount of intra-LUT flexibility is given in Equation (4). As each LUT has 4 inputs, the upper bound of intra-LUT flexibility is 48 (for $l$=2 or 3). In comparison, each CLB has 10 LUTs that are fully utilized in most cases. Therefore the amount of inter-LUT flexibility is 10! according to Equation (5). Enumerating all these possibilities for each CLB is infeasible, since a design may include a large number of CLBs.

To efficiently exploit inter-LUT flexibility, we propose a bipartite graph based approach. First, an $m \times n$ matrix $RR_{LUT}$ is used to record the reconfiguration cost between the $m$ LUTs in the new content and the $n$ LUTs in the original content of a CLB. Each entry $RR_{LUT}[i][j]$ is the minimum cost of configuring logical $LUT_i$ to physical $LUT_j$. This minimum cost is obtained by exploiting the intra-LUT flexibility of all the possible input combinations.

Once the entire $RR_{LUT}$ matrix is computed, finding the best mapping that minimizes the reconfiguration cost of the CLB can be converted to a *weighted bipartite matching* problem. The logical LUTs and physical LUTs are represented as the two sets of nodes in the bipartite graph, while $RR_{LUT}[i][j]$ is the weight of the edge from node $i$ on one side to node $j$ on the other side. The best match between these two sets of LUTs is the one that minimizes the sum of the weights of all the matched edges. It can be obtained using the classical *Kuhn-Munkres* (KM) algorithm [29]. This algorithm takes $RR_{LUT}$ as inputs, maintains a *weight label* for each node, iteratively searches for *augmented paths* that minimize the sum of weight labels, and finally outputs the best match and the lowest matching score. In this way, both the intra-LUT and inter-LUT flexibilities are exploited to their maximum extent.

As a concrete example, consider the computation of the minimum cost for configuring $V2$ to $U1$ in Figure 3(b). The content of matrix $RR_{LUT}[2][2]$ is $\{3, 1, 2, 2\}$. Then the KM algorithm identifies the best match of $\{LUT2 \rightarrow LUT1, LUT1 \rightarrow LUT2\}$ and the corresponding minimum score of $1 + 2 = 3$.

Overall, the complexity for constructing the $RR_{LUT}$ matrix is $O(mn)$, while the complexity of the KM algorithm is $O(n^3)$. Clearly, this algorithm is much more efficient than enumerating all the $n!$ matching possibilities. Since this procedure is performed to each CLB in the new design, the overall complexity of the OO placement algorithm is $O(un^3)$, assuming that the new design includes a total number of $u$ CLBs and each CLB includes $n$ LUTs.

### B. Min-bit-flip (MBF) Placement

The MBF placement solely focuses on reducing the CLB reconfiguration cost. Timing and routing constraints are only considered at the routing stage of VTR. For each CLB, the MBF placement exploits intra-LUT and inter-LUT flexibilities in the same way as the OO placement. However, it does not adopt the VTR placement algorithm, but constructs a bipartite graph model to fully exploit inter-CLB reconfiguration flexibility.

Inter-CLB flexibility can be exploited in the same way as inter-LUT flexibility. However, inter-CLB flexibility is much more complex, since even a small design may include hundreds of CLBs. Using the aforementioned bipartite graph based approach, a $u \times v$ matrix $RR_{CLB}$ can record the reconfiguration cost between all the $u$ CLBs in the new design and all the $v$ blocks in the specified FPGA area. Each entry $RR_{CLB}[i][j]$ is the minimum cost of configuring $CLB_i$ in the new design to $BLK_j$ in FPGA's original content. Similarly, this minimum cost is the lowest among all possible permutations of intra-LUT and inter-LUT flexibilities, which can be obtained in the way described in Section V-A.

Once the entire $RR_{CLB}$ matrix is computed, the next step is to identify the MBF placement. Again, a bipartite graph with $u + v$ nodes and $u \times v$ edges is constructed. CLBs and FLGA blocks are respectively represented as the two sets of nodes, while $RR_{CLB}[i][j]$ is the weight of the edge from node $i$ to node $j$. The best match between these two sets of nodes can also be obtained using the KM algorithm [29]. If, in the best match, $CLB_i$ is mapped to $BLK_j$, it inherits the position of $BLK_j$. Overall, the MBF placement has a complexity of $O(uv^2)$ and is able to exploit all the three types of flexibilities to their maximum extent.

### C. Reconfiguration & Timing balanced (R&T) Placement

The goal of the R&T placement is to balance reconfiguration cost and timing during the VTR placement process. It uses the MBF placement as the initial placement and applies VTR optimizations on it. Overall, the R&T placement is highly compatible to the existing VTR placement algorithm and has the same complexity as it.

A brief introduction of the original VTR placement algorithm is useful for highlighting the changes introduced by the proposed R&T placement. VTR starts from a randomly generated initial placement. At every iteration, it randomly selects two CLBs and swaps them. A cost function that combines wire length and timing delay is used to evaluate each swap decision. A swap is definitely accepted if it is beneficial and probabilistically accepted even if it increases the cost function.

In comparison, the proposed R&T placement algorithm adopts the same flow and selects CLBs for swapping in the same way as VTR. However it makes three major changes to VTR: the initial placement, the cost function, and the condition for accepting swap decisions. The overall algorithm is shown in Algorithm 1, while the major changes are described below.

**Algorithm 1** Reconfiguration & Timing balanced (R&T) placement

**Input:** Netlist, MBF placement $P$, Matrix $RR_{CLB}$, threshold $\alpha$
**Output:** Positions of each CLB

Compute initial $bb\_cost$, $td\_cost$, $rr\_cost$;
Initial reconfiguration weight $W_{rr} \Leftarrow 0$;
$initial\_budget \Leftarrow \alpha \times (OOCost - MBFCost)$;
$remaining\_budget \Leftarrow initial\_budget$;
**while** $Exit() = 0$ **do**
  **for** $Iter \Leftarrow 0$ to Inner_Loop_Limit increase by 1 **do**
    $P_{new} \Leftarrow$ RandomSwapTwoCLBs();
    $\Delta bb\_cost \Leftarrow bb\_cost(P_{new}) - bb\_cost(P)$;
    $\Delta td\_cost \Leftarrow td\_cost(P_{new}) - td\_cost(P)$;
    $\Delta rr\_cost \Leftarrow rr\_cost(P_{new}) - rr\_cost(P)$;
    Normalize $\Delta bb\_cost$, $\Delta td\_cost$, $\Delta rr\_cost$;
    $\Delta Cost \Leftarrow W_{rr} \times \Delta rr\_cost + (1 - W_{rr}) \times \Delta td\_cost$;
    $\Delta Cost \Leftarrow W_{bb} \times \Delta bb\_cost + (1 - W_{bb}) \times \Delta Cost$;
    **if** $\Delta Cost < 0$ **then**
      Accept; $P \Leftarrow P_{new}$;
      Update $bb\_cost, td\_cost, rr\_cost$;
    **else**
      Reject;
    **end if**
    $remaining\_budget \Leftarrow remaining\_budget - \Delta rr\_cost$;
  **end for**
  **if** $remaining\_budget \leq 0$ **then**
    $W_{rr} \Leftarrow 1$;
  **else**
    $W_{rr} \Leftarrow 1 - remaining\_budget/initial\_budget$;
  **end if**
**end while**

First, instead of starting from a randomly generated placement, the R&T algorithm takes MBF placement as the initial placement. Since MBF has minimum CLB reconfiguration cost, using it as the starting point is promising for achieving a low reconfiguration cost together with good timing characteristics.

Second, the cost function of VTR placement is augmented to include reconfiguration cost, in addition to traditional metrics such as wire length and critical path delay, as a factor to guide CLB swapping. A user-specified threshold is used to balance reconfiguration cost and

timing delay during the placement process. These changes can be observed in Algorithm 1. At Line 12, a weight $W_{rr}$ is used to balance reconfiguration cost, denoted as $rr\_cost$, and timing delay cost, denoted as $td\_cost$. Then, at Line 13, a weight $W_{bb}$ is used to balance bounding box cost, denoted as $bb\_cost$ (used to estimate wire length), and $\Delta Cost$. Furthermore, to gradually adjust the weights of CLB reconfiguration cost and critical path delay, R&T uses a fixed $W_{bb}$ but dynamically updates $W_{rr}$ based on a user-specified threshold $\alpha$. At Line 3, an $initial\_budget$ is computed based on $\alpha$ and the difference between the CLB reconfiguration costs achieved by MBF placement and OO placement. The larger the $\alpha$, the higher the $initial\_budget$, and hence the lower the weight of reconfiguration cost. Meanwhile, a variable called $remaining\_budget$ is used to record the amount of budget left during placement. At each iteration, $W_{rr}$ is updated based on the difference between these two budgets at Line 25. At the beginning, $remaining\_budget$ equals $initial\_budget$ (Line 4) and therefore $W_{rr}$ is close to 0, resulting a higher weight being given to timing delay cost ($td\_cost$) at Line 12. As more budget is consumed (Line 20), the weight of reconfiguration cost ($W_{rr}$) increases. Finally, when no budget is left, $W_{rr}$ is set to 1 at Line 23 and hence the cost function does not consider $td\_cost$ but only $bb\_cost$ and $rr\_cost$ to evaluate CLB swap decisions.

Finally, the R&T algorithm eliminates the randomness in the acceptance condition of swap decisions in VTR placement. Specifically, the original VTR algorithm uses an annealing method [11] that probabilistically accepts a swap decision even if it increases the cost (i.e., $\Delta Cost$ is larger than 0). However, such randomness is eliminated in the proposed algorithm. Since R&T uses the min-bit-flip placement as the initial placement, CLB swapping is accepted only if it reduces the cost (Lines 14-19).

## VI. Experimental Evaluation

This section evaluates the proposed reconfiguration optimization approaches in terms of their effectiveness in reducing reconfiguration cost and their influence on design timing.

### A. Experimental Setup

The experimental FPGA architecture is *k06n10* (based on Altera Stratix IV) with a 45nm fabrication library and an area-delay model offered by VTR 7.0 CAD flow [11]. Each CLB consists of 10 BLEs, as shown in Figure 1. Each LUT has 4 inputs, while each SB has a channel width of 60 and each channel has a fanout of 3. The fraction of tracks in a channel to which each pin connects is 0.15. Routing is timing driven.

Six different placement results are compared in our studies. They are listed in Table II together with the different configurations used. The *Baseline* is original VTR placement optimized for timing, while the *Optimized Original* (OO) performs intra-LUT and inter-LUT reconfiguration optimizations on top of the baseline. The *min-bit-flip* (MBF) placement minimizes CLB reconfiguration cost in the way described in Section V-B, while the *Reconfiguration & Timing balanced* (R&T) placement described in Section V-C is tested with three different values of threshold $\alpha$. As Algorithm 1 shows, $\alpha$ determines the condition for triggering reconfiguration-driven cost functions during VTR placement. The larger the value of $\alpha$, the lower the weight of reconfiguration cost.

We evaluate these algorithms with standard MCNC benchmark circuits [30] offered by VTR to ensure that the obtained results are representative and convincing. Table III shows the 9 benchmarks used in the experiment, sorted in the ascending order of their sizes. For each benchmark, its numbers of LUTs, CLBs, and SBs and the baseline CLB and SB configuration costs are shown. To ensure

Table II. Six placements and their configurations

| Schemes | Flexibilities Exploited | | | Timing Optimization |
|---|---|---|---|---|
| | intra-LUT | inter-LUT | inter-CLB | |
| Baseline | - | - | - | √ |
| Optimized Original | √ | √ | - | √ |
| Min-Bit-Flip | √ | √ | √ | - |
| R&T1 ($\alpha = 0.25$) | √ | √ | √ | √ |
| R&T2 ($\alpha = 0.50$) | √ | √ | √ | √ |
| R&T3 ($\alpha = 0.75$) | √ | √ | √ | √ |

Table III. Benchmark Circuits

| No | Benchmark | LUT# | CLB# | CLB Cost | SB# | SB Cost |
|---|---|---|---|---|---|---|
| 1 | tseng | 1046 | 105 | 12958 | 196 | 4302 |
| 2 | ex5p | 1064 | 107 | 14764 | 196 | 4588 |
| 3 | diffeq | 1494 | 150 | 18791 | 256 | 4972 |
| 4 | alu4 | 1522 | 153 | 19460 | 256 | 5128 |
| 5 | seq | 1750 | 175 | 21780 | 289 | 6599 |
| 6 | s298 | 1930 | 194 | 25520 | 324 | 8512 |
| 7 | elliptic | 3602 | 361 | 44372 | 576 | 18391 |
| 8 | spla | 3690 | 369 | 51268 | 576 | 19875 |
| 9 | ex1010 | 4598 | 460 | 55768 | 676 | 18406 |

fairness in comparison, redundant write elimination [26] is applied to both the baseline and the proposed approaches. We have performed 8 evaluation cases: Case $i$ corresponds to mapping the $i^{th}$ circuit to the $(i+1)^{th}$ one, assuming that $i$ is the design to be configured and $i+1$ is the design currently implemented on the NVM-based FPGA.

The results are shown in four metrics: CLB reconfiguration cost, total reconfiguration cost, critical path delay,and algorithm execution time. We do not report design area since all the proposed algorithms fulfill the given area constraint and impose no area overhead.

### B. Reconfiguration Cost

Figure 5 shows CLB reconfiguration costs of "OO", "MBF", and "R&T" with three different configurations, normalized to baseline (e.g., a cost of 40% means the algorithm reduces baseline reconfiguration cost by 60%). These values reflect the direct effect of the proposed placement algorithms, as all of them target to reduce the CLB reconfiguration cost. Overall, the proposed algorithms are quite effective. "OO" exploits two types of flexibilities and reduces 73.2% of the CLB reconfiguration cost on average. As expected, "MBF" delivers the maximum amount of average reduction of 83.7%, as it exploits all three types of flexibilities. The three "R&T" placements respectively reduce the CLB reconfiguration cost by 81.5%, 81.4% and 81.3%, which are close to the upper bound given by "MBF". As $\alpha$ increases, in some cases the CLB reconfiguration cost does not increase as expected. This is due to the large amount of randomness in the original VTR placement algorithm, which randomly selects two CLBs for swapping at each iteration.

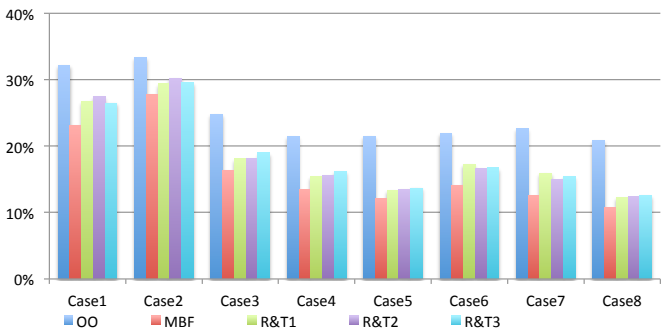


Figure 5. CLB reconfiguration cost of different schemes. Case $i$ corresponds to mapping the $i^{th}$ circuit in Table III to the $(i+1)^{th}$ one.

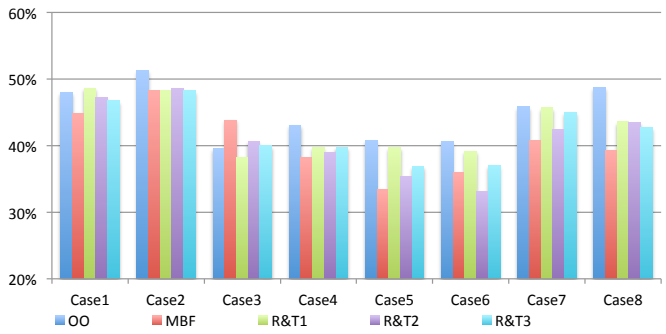


Figure 6. Total reconfiguration cost of different schemes. Case $i$ corresponds to mapping the $i^{th}$ circuit in Table III to the $(i+1)^{th}$ one.

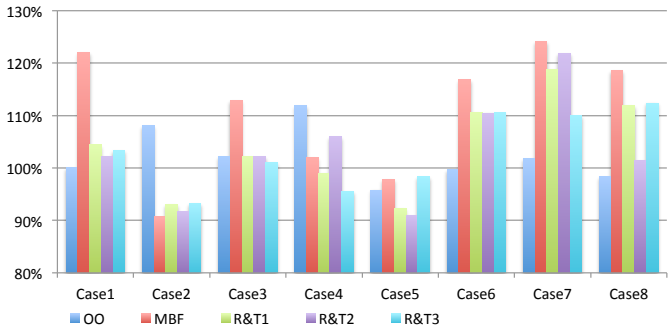


Figure 7. Normalized critical path length of different schemes. Case $i$ corresponds to mapping the $i^{th}$ circuit in Table III to the $(i+1)^{th}$ one.

Figure 6 shows the total reconfiguration cost, that is, the number of NVM cells written to program all CLBs and SBs. These values show both the direct and the secondary effects of the proposed placement algorithms. Taking "MBF" as an example, it optimizes CLB placement without imposing any timing or routing constraint. As a result, it may require more tracks or even more SBs to route the design, thus increasing the SB reconfiguration cost. As a result, "MBF" outperforms all the other placement algorithms in reducing the CLB reconfiguration cost shown in Figure 5, but not the total reconfiguration cost for Cases 3, and 6 in Figure 6. Overall, "OO", "MBF" and the three "R&T" schemes reduce the total reconfiguration cost by 56.3%, 59.4%, 57.2%, 58.8% and 58.0%, respectively. Comparison of the three "R&T" schemes confirms that adopting a smaller value of $\alpha$ can effectively reduce the reconfiguration cost.

### C. Timing and Routing

In addition to reconfiguration costs, critical path delay is also reported since all the proposed placement algorithms share the same goal of reducing reconfiguration cost within acceptable degradation of critical path delay. The reference baseline adopts timing-driven VTR placement and routing. Delay values of the five proposed algorithms, collected at the post-routing stage and normalized to baseline, are shown in Figure 7.

The "OO" placement influences critical path delay as it changes pin positions of each CLB and hence the performance of the router. Figure 7 shows that "OO" slightly increases the critical path delay for Cases 2, 3, and 4, while for other cases it reduces the delay. On average its critical path is 2% longer than the baseline. In comparison, "MBF" and "R&T" have a larger impact on critical paths as they change both CLB positions and pin positions of each CLB. On average, the critical path of "MBF" is 11% longer than the baseline, while the critical paths of R&T1, R&T2 and R&T3 are 4%, 3%, and 3% longer, respectively. These results confirm the ability of the "R&T" algorithm in balancing reconfiguration cost and timing delay.

Table IV. RUNNING TIME FOR DIFFERENT STAGES

| | | Baseline | | MBF | | R&T | |
|---|---|---|---|---|---|---|---|
| | KM (ms) | P (s) | R (s) | P (s) | R (s) | P (s) | R (s) |
| Case1 | 0.514 | 6.430 | 6.807 | 0.530 | 4.363 | 2.907 | 3.744 |
| Case2 | 1.305 | 6.382 | 6.643 | 0.527 | 12.019 | 2.905 | 8.964 |
| Case3 | 2.298 | 10.23 | 4.294 | 0.759 | 10.928 | 4.500 | 14.090 |
| Case4 | 1.754 | 9.083 | 12.37 | 0.716 | 8.253 | 4.358 | 10.224 |
| Case5 | 2.967 | 13.87 | 22.13 | 0.811 | 17.032 | 6.612 | 18.270 |
| Case6 | 2.318 | 13.77 | 15.21 | 0.905 | 9.574 | 6.358 | 24.305 |
| Case7 | 5.980 | 39.93 | 34.54 | 1.794 | 52.812 | 18.964 | 52.482 |
| Case8 | 18.08 | 42.70 | 88.18 | 1.750 | 55.817 | 20.324 | 115.540 |

### D. Algorithm Execution Time

To evaluate the complexity of the proposed placement algorithms, Table IV reports the latency of the KM algorithm [29] in searching for the best positions of CLBs, as well as the latencies of the VTR placement (P) and routing (R) processes. Here the baseline, "MBF", and "R&T" schemes are reported while the "OO" scheme is omitted since it has the same latency as the baseline.

The KM algorithm only imposes negligible overhead in execution, since it finishes execution in milliseconds while both VTR placement and routing processes are in the scale of seconds. Incorporating this algorithm into VTR does not degrade the synthesis speed.

Since "MBF" performs placement directly based on the results of bipartite graph matching, it has much shorter execution time than the baseline. On average, "MBF" placement is 18 times faster. Regarding the latency of routing, "MBF" is comparable to the baseline, although in some cases it requires more routing iterations. On average, the routing latency of "MBF" is 7% longer than the baseline.

The "R&T" placement algorithm has an execution time about 50% shorter than the baseline placement. This is because "R&T" adopts a greedy strategy to accept only good swap decisions, resulting in the placement process to converge more quickly than the baseline that probabilistically accepts bad swap decisions. On the other hand, the greedy strategy together with the new cost function slightly degrades the timing and routability of the placement result. As a result, "R&T" tends to require more iterations to search for a high-quality routing result, thus leading to a routing process 22% longer than the baseline.

Overall, the four sets of results show that the proposed placement algorithms are able to fulfill area constraints and significantly reduce the cost for (re)configuring a design on an NVM-based FPGA, without imposing noticeable overhead on critical path delay or algorithm execution time. They therefore bridge the gap between NVM-based FPGAs and current FPGA synthesis techniques, and hence make NVM-based FPGAs more attractive to future self-adaptive embedded systems.

## VII. CONCLUSIONS

In this paper, we have proposed a set of (re)configuration optimization techniques for NVM-based FPGAs, aiming at mitigating the adverse impact on (re)configuration caused by the slow and costly NVM write operations. A mathematical model that characterizes CLB reconfiguration costs at the bit level has been proposed. Then, three types of flexibilities that can be exploited to reduce the reconfiguration cost have been identified. Finally, three placement approaches that exploit these flexibilities in different ways have been proposed, allowing designers to fine-tune the placement process to balance the reconfiguration cost and traditional timing and routability constraints. VTR-based experimental studies show that the proposed

optimizations can reduce the total reconfiguration cost by up to 67% with a 4.6% average degradation in critical path delay. By speeding up reconfiguration and prolonging lifetime of NVM-based FPGAs, the proposed framework makes NVM-based FPGAs attractive to future self-adaptive embedded systems.

## REFERENCES


[1] K. Huang, Y. Ha, R. Zhao, A. Kumar, and Y. Lian, "A low active leakage and high reliability phase change memory (PCM) based non-volatile FPGA storage element," *IEEE Transactions on Circuits and Systems*, vol. 61, 2014.

[2] H. Asadi and M. Tahoori, "Analytical techniques for soft error rate modeling and mitigation of FPGA-based designs," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 15, 2007.

[3] P. Graham, M. Caffrey, J. Zimmerman, P. Sundararajan, E. Johnson, and C. Patterson, *Consequences and categories of SRAM FPGA configuration SEUs*, Los Alamos National Laboratory and Xilinx, Sep. 2003.

[4] *Understanding soft and firm errors in semiconductor devices*, Actel, Dec. 2002.

[5] K. J. Han, N. Chan, S. Kim, B. Leung, V. Hecht, and B. Cronquist, "A novel flash-based FPGA technology with deep trench isolation," in *Non-Volatile Semiconductor Memory Workshop*, 2007, pp. 32–33.

[6] P. Gaillardon, M. Ben-Jamaa, G. Beneventi, F. Clermidy, and L. Perniola, "Emerging memory technologies for reconfigurable routing in FPGA architecture," in *International Conference on Electronics, Circuits, and Systems (ICECS)*, 2010, pp. 62–65.

[7] W. Zhao, E. Belhaire, V. Javerliac, C. Chappert, and B. Dieny, "Evaluation of a non-volatile FPGA based on MRAM technology," in *International Conference on Integrated Circuit Design and Technology (ICICDT)*, 2006, pp. 1–4.

[8] Y. Chen, J. Zhao, and Y. Xie, "3D-NonFAR: Three-dimensional non-volatile FPGA architecture using phase change memory," in *International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010, pp. 55–60.

[9] *International technology roadmap for semiconductors*, 2013.

[10] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, 2008.

[11] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, 2014.

[12] *Partial reconfiguration user guide*, Xilinx, Mar. 2011.

[13] R. He, Y. Ma, K. Zhao, and J. Bian, "ISBA: an independent set-based algorithm for automated partial reconfiguration module generation," in *International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 500–507.

[14] S. Raaijmakers and S. Wong, "Run-time partial reconfiguration for removal, placement and routing on the Virtex-II Pro," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pp. 679–683.

[15] Z. Yang and G. Choi, "Reconfigurable multi-functioning logic structures: a case study of MMX/floating-point unit design," in *Computer Society Workshop On VLSI*, 1999, pp. 76–81.

[16] F. Mehdipour, M. Zamani, H. Ahmadifar, M. Sedighi, and K. Murakami, "Reducing reconfiguration time of reconfigurable computing systems in integrated temporal partitioning and physical design framework," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2006, pp. 308–315.

[17] S. Wilton, "Architectures and algorithms for field programmable gate arrays with embedded memory," *Ph.D. dissertation, University of Toronto*, 1997.

[18] M. Rullmann and R. Merker, "A reconfiguration aware circuit mapper for FPGAs," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2007, pp. 1–8.

[19] X. Shi, D. Zeng, Y. Hu, G. Lin, and O. Zaiane, "Enhancement of incremental design for FPGAs using circuit similarity," in *International Symposium on Quality Electronic Design (ISQED)*, 2011, pp. 1–8.

[20] A. Ling, S. Brown, J. Zhu, and S. Safarpour, "Towards automated ECOs in FPGAs," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2009, pp. 3–12.

[21] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "DeltaSyn: An efficient logic difference optimizer for ECO synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, 2009, pp. 789–796.

[22] D. Ziener, S. Assmus, and J. Teich, "Identifying FPGA IP-cores based on lookup table content analysis," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1–6.

[23] H. A. Khouzani, Y. Xue, and C. Yang, "Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2014, pp. 327–330.

[24] H. A. Khouzani, C. Yang, and J. Hu, "Leveraging microarchitectural side channel information to efficiently enhance program control flow integrity," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014, pp. 1–9.

[25] C. Liu, C. Yang, and Y. Shen, "Improving performance and lifetime of DRAM-PCM hybrid main memory through a proactive page allocation strategy," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015, pp. 508–513.

[26] A. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse, "Increasing PCM main memory lifetime," in *Design, Automation and Test in Europe (DATE)*, 2010, pp. 914–919.

[27] C. Liu and C. Yang, "Improving multilevel PCM reliability through age-aware reading and writing strategies," in *'International Conference on Computer Design (ICCD)*, 2014, pp. 264–269.

[28] M. Zhao, Y. Xue, C. Yang, and C. Xue, "Minimizing MLC PCM write energy for free through profiling-based state remapping," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015, pp. 502–507.

[29] H. Kuhn, "The hungarian method for the assignment problem," in *Naval Research Logistics Quarterly*, 1955, pp. 29–47.

[30] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *International Symposium on Circuits and Systems (ISCAS)*, 1989, pp. 1929–1934.