A Hybrid Nano/CMOS Dynamically Reconfigurable System—Part II: Design Optimization Flow

13

WEI ZHANG and NIRAJ K. JHA
Princeton University
and
LI SHANG
University of Colorado, Boulder

In Part I of this work, a hybrid nano/CMOS reconfigurable architecture, called NATURE, was described. It is composed of CMOS reconfigurable logic and interconnect fabric, and nonvolatile nano on-chip memory. Through its support for cycle-by-cycle runtime reconfiguration and a highly-efficient computation model, temporal logic folding, NATURE improves logic density and area-delay product by more than an order of magnitude compared to existing CMOS-based field-programmable gate arrays (FPGAs). NATURE can be fabricated using mainstream photo-lithography fabrication techniques. Thus, it offers a currently commercially feasible architecture with high performance, superior logic density, and excellent runtime design flexibility.

In Part II of this work, we present an integrated design and optimization flow for NATURE, called NanoMap. Given an input design specified in register-transfer level (RTL) and/or gate-level VHDL, NanoMap optimizes and implements the design on NATURE through logic mapping, temporal clustering, temporal placement, and routing. As opposed to other design tools for traditional FPGAs, NanoMap supports and leverages temporal logic folding by integrating novel mapping techniques. It can automatically explore and identify the best temporal logic folding configuration, targeting area, delay or area-delay product optimization. A force-directed scheduling technique is used to optimize and balance resource usage across different folding cycles. By supporting logic folding, NanoMap can provide significant design flexibility in performing area-delay trade-offs under various user-specified constraints. We present details of the mapping procedure and results for different architectural instances. Experimental results demonstrate that NanoMap can judiciously trade off area and delay targeting different optimization goals, and effectively exploit the advantages of NATURE.

Part I of this work will appear in JETC Vol. 5, No. 4.

This work was supported by NSF under Grant no. CNS-0719936.
Authors' addresses: W. Zhang, N. K. Jha, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544. L. Shang, Department of Electrical and Computer Engineering, University of Colorado at Boulder, CO 80305.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1550-4832/2009/08-ART13 $10.00
DOI 10.1145/1568485.1568487 http://doi.acm.org/10.1145/1568485.1568487

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Dynamic reconfiguration, design optimization flow, logic folding, NATURE

ACM Reference Format:

Zhang, W., Jha, N. K., and Shang, L. 2009. A Hybrid Nano/CMOS Dynamically Reconfigurable System—Part II: Design Optimization Flow. ACM J. Emerg. Technol. Comput. Syst. 5, 3, Article 13 (August 2009), 31 pages. DOI = 10.1145/1568485.1568487
http://doi.acm.org/10.1145/1568485.1568487

1. INTRODUCTION

Technology scaling has been the primary driving force behind integrated circuits for several decades. As projected by the International Technology Roadmap for Semiconductors [ITRS 2007], CMOS technology is approaching its physical limits. Hence, tremendous efforts are being devoted to nanoscale device and fabrication research [Cui et al. 2003; Javey et al. 2004; Zhang and Jha 2005]. As discussed in Part I, recent research on reliable nanoscale circuits and architectures has resulted in a variety of nanoelectronic and hybrid nano/CMOS reconfigurable designs [DeHon 2006; Snider et al. 2004; Goldstein and Budiu 2001; Luo 2002; Strukov and Likharev 2005; Rad and Tehranipoor 2006]. These designs demonstrate significant improvement in performance and integration density compared to existing CMOS-based solutions. However, they require a self-assembly fabrication process, which is unlikely to be mature in the near future.

In Part I, we proposed a hybrid nano/CMOS reconfigurable architecture, called NATURE, which can be fabricated using a CMOS-compatible photolithography fabrication process. It addresses two primary challenges in existing CMOS-based FPGAs: logic density and efficiency of runtime reconfiguration. By using nonvolatile high-performance, high-density nano RAMs, such as carbon nanotube-based RAM [Nantero 2008], magnetoresistive RAM [Tehrani et al. 2003] or phase-change memory [Lai 2003], as on-chip reconfiguration storage, NATURE improves the logic density of the FPGA by more than an order of magnitude. Since the access latency of on-chip storage is small, it enables fine-grain cycle-level dynamic reconfiguration. This leads to support for temporal logic folding, which was introduced in Part I. A large logic circuit can be partitioned into a sequence of logic stages and the configuration bits for these stages can be stored in the on-chip configuration memory. At runtime, stage-by-stage, the logic circuit can be configured into the same hardware through fine-grain dynamic reconfiguration and executed in different clock cycles. Moreover, due to the nonvolatile property of nano RAMs, reconfiguration bits can be maintained in it even when the power supply is switched off. Hence, there is no need to repeatedly load the reconfiguration bits from off-chip memory. This reduces the power consumption and helps with secure processing.

In this article, we present NanoMap, an integrated design optimization platform for NATURE. NanoMap conducts design optimization from the RTL down

to the physical level. Preliminary work on NanoMap is presented in [Zhang et al. 2007]. In this article, we improve the integrated flow to include more mapping scenarios and present further details. Power consumption for large benchmark implementations in NATURE is also analyzed. In the past, numerous design tools have been proposed for reconfigurable architectures with various optimization targets, including area/delay minimization and routability maximization (see Cong [1996] for an excellent survey). They mainly perform technology mapping from Boolean network to netlist of look-up tables. Our approach differs fundamentally from previous work by supporting the temporal logic folding model and targeting the design flexibility enabled by logic folding. Given an input design specified in RTL and/or gate-level netlist (netlist of look-up tables from the output of technology mapping), NanoMap further optimizes and implements the design on NATURE through logic mapping, temporal clustering, temporal placement, and routing. Given user-specified area/delay constraints, the mapper can automatically explore and identify the best logic folding configuration based on the chosen instance of the NATURE architecture family and make appropriate trade-offs between delay and area efficiency. It uses a force-directed scheduling (FDS) technique [Paulin and Knight 1989] to balance resource usage across different logic folding cycles. Traditional place-and-route methods are modified to integrate the temporal logic folding model. Combining NanoMap with existing commercial synthesis tools, such as Synopsys Design Compiler [Synopsys 2009], provides a complete design automation flow for NATURE.

This work makes the following contributions:

—It includes an integrated design automation flow to enable efficient use of NATURE. It provides support for fine-grain temporal logic folding and allows flexible trade-off between area efficiency and performance.

—It contains a logic mapper that automatically identifies the best temporal logic folding configuration and uses force-directed techniques to yield the best area efficiency.

—It contains an interface to commercial architectural synthesis flow. Design inputs support RTL, gate-level, or mixed descriptions.

—It also provides opportunities to evaluate the efficacy of NATURE for implementing large designs. This helps point to new research directions to further optimize the NATURE architecture.

The article is organized as follows. A motivational example is used to illustrate the design flow in Section 2. The detailed flow of NanoMap is discussed in Section 3. The logic mapping, temporal clustering, and temporal place-and-route steps are discussed in Section 4, 5, and 6, respectively. Experimental results are presented in Section 7. Conclusions are given in Section 8.

2. MOTIVATING EXAMPLE

In this section, we use an RTL example to demonstrate the design optimization flow of NanoMap, that is, how NanoMap maps the input design to NATURE.

First, we introduce some concepts for ease of exposition. Given a circuit, such as the one shown in Figure 1(a), the registers contained in it are first levelized.

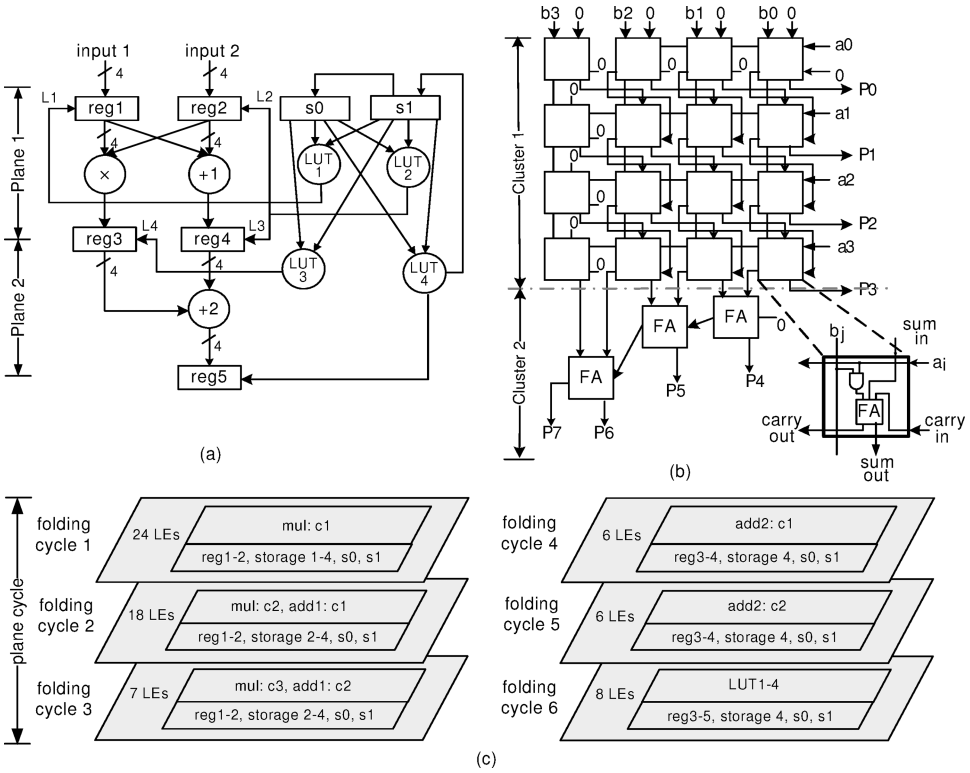

Fig. 1. Illustration of (a) an example circuit (b) module partition (c) mapping result.

The logic between two levels of registers is referred to as a *plane*. For instance, the example circuit contains two planes. The registers associated with the plane are called *plane registers*. The propagation cycle of a plane is called the *plane cycle*. Using temporal logic folding, each plane is further partitioned into *folding stages*. Resources can be shared among different folding stages within a plane or across planes. The propagation cycle of a single folding stage is defined as a *folding cycle*. Note that different planes should consist of the same number of folding stages to guarantee global synchronization. Thus, the key issue is to determine how many planes are folded together and the appropriate folding level, that is, the number of folding stages in one plane, to achieve the best area-delay trade-off under specified design constraints.

The example shown in Figure 1(a) comprises a four-bit controller-datapath. An adder and a multiplier are included in the first plane and a single adder in the second plane. The controller consists of flip-flops s0 and s1 and LUTs LUT1–LUT4 that control loading of registers reg1–reg5 in the datapath. Assuming ripple-carry adders and parallel multipliers, the circuit requires 58 LUTs in all (50 LUTs for plane 1 and 8 LUTs for plane 2) and 22 flip-flops. The adder consists of eight LUTs with a logic depth, that is, the number of LUTs along the critical path, of four. The multiplier consists of 38 LUTs with a logic depth of seven. Hence, the logic depth is seven for plane 1 and four for plane 2. The mapping

procedure can target many different optimization objectives. For the example, suppose the objective is to minimize delay under a total area constraint of 30 logic elements (LEs). Here, delay indicates the register input to register output delay. We assume each LE contains one LUT and two flip-flops. Hence, 30 LEs contain 30 LUTs along with 60 flip-flops. Since the number of available flip-flops is more than required, we concentrate on the LUT constraint.

NanoMap uses an iterative optimization flow. It first chooses an initial folding level based on the constraints and input design structure and then gradually adjusts the folding level to achieve the optimization goal. As discussed in Part I of the article, fewer folding stages generally lead to better delay. Thus, NanoMap can start with a folding level that results in a minimal number of folding stages. In the example, since the design allows cross-plane folding (since it is not pipelined), we can fold one plane on top of the other. The minimal number of folding stages in each plane is equal to the maximum number of LUTs among all the planes divided by the LUT constraint: $\lceil \frac{50}{30} \rceil = 2$, that is, at least two folding stages are required to meet the LUT constraint. Since there are two planes, two folding stages are needed for each plane. The initial folding level is then obtained by the maximum logic depth among all the planes divided by the number of folding stages, which equals $\lceil \frac{7}{2} \rceil = 4$.

Next, based on the chosen folding level, the adder and multiplier are partitioned into a series of connected LUT clusters in such a way that if the folding level is p, then all the LUTs at a depth less than or equal to p in the module are grouped into the first cluster, all the LUTs at a depth larger than p but less than or equal to $2p$ are grouped into the second cluster, and so on. The LUT cluster can be considered in its entirety and contained in one folding stage. By dealing with LUT clusters instead of a group of single LUTs, the logic mapping procedure can be greatly sped up. Figure 1(b) shows the partition for the multiplier using level-4 folding. However, the first LUT cluster of the multiplier needs 32 LUTs, already exceeding the area constraint. This is because the logic is not balanced well between two planes. Thus, the folding level has to be further decreased to level-3 in order to guarantee that each LUT cluster can be accommodated within the available LEs. Correspondingly, the number of folding stages increases to $\lceil \frac{7}{3} \rceil = 3$ for each plane.

Next, FDS is used to determine the folding cycle assignment of each LUT and LUT cluster to balance the resource usage across the three folding stages in each plane. If the number of LUTs and flip-flops required by every folding stage is below the area constraint, that is, 30 LEs, the solution is valid and offers the best possible delay. Otherwise, the folding level is reduced by one, followed by another round of optimization until the area constraint is met, assuming it can be satisfied.

Figure 1(c) shows the mapping result for level-3 folding for the six folding stages in two planes. Note that plane registers, which provide inputs to the plane, need to exist through all the folding stages in the plane. The first folding cycle requires 24 LEs, which is the most number of LEs among the six folding stages. They are required for mapping the first cluster of the multiplier ($mul{:}c1$). Four-bit registers reg1-reg2, LUT computation results for LUT1-LUT4 and one-bit state registers s0 and s1 are mapped to the available flip-flops inside the

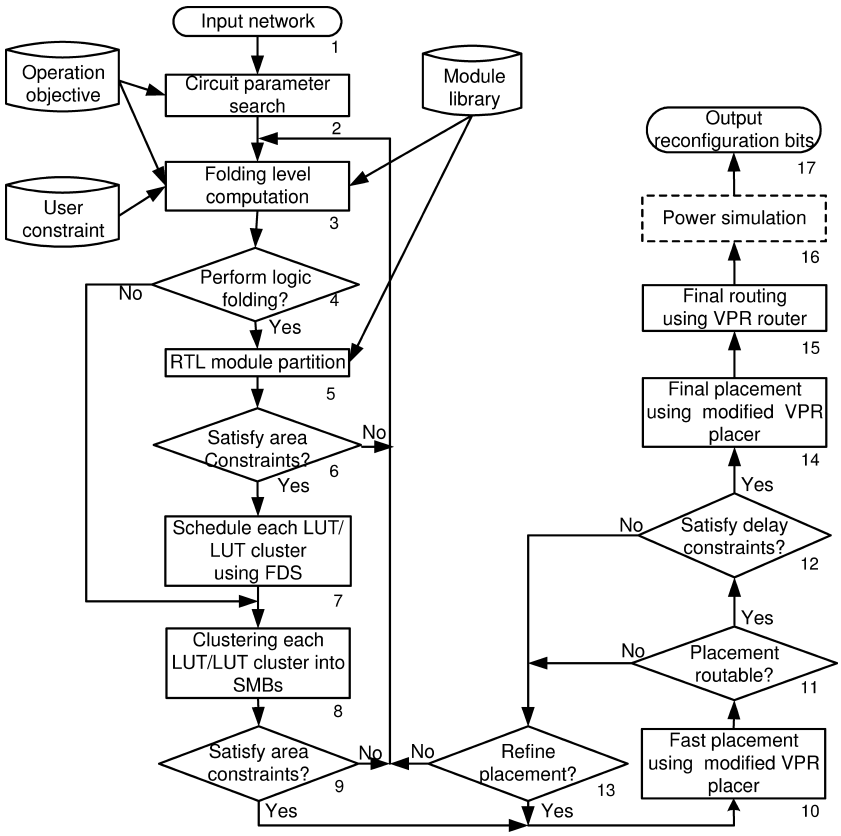

Fig. 2. Automatic design flow.

LEs assigned to the multiplier. The results from the LUT computations are also stored in the available flip-flops inside the LEs. Note that the computation result of LUT1 is not needed after folding cycle 1, and hence not kept. reg3-reg5 are not created until they are used in order to save area. From the figure, we can see that 18, 7, 6, 6, and 8 LEs are needed for the following folding cycles, respectively. Hence, the number of LEs for mapping this RTL circuit is the maximum required across all the folding cycles, that is, 24, which is less than the area constraint.

Next, in each folding stage, clustering, which groups LEs into super macroblocks (SMBs), placement and routing are performed to produce the final layout of the implementation.

3. NANOMAP: OVERVIEW OF THE OPTIMIZATION FLOW

In this section, we present the steps of NanoMap, an integrated design optimization flow developed for NATURE in detail and discuss its complexity. Figure 2 illustrates the complete flow. Given an RTL or gate-level design, NanoMap performs logic mapping, temporal clustering, temporal placement and routing, and produces the configuration bitmap for NATURE. If the power consumption of

the design needs to be evaluated, a power simulation (Step 16) can also be performed according to the method described in Part I of the article.

—*Logic Mapping.* (Steps 2–7). These steps use an iterative approach to identify the best folding level based on user-specified design constraints, optimization objectives, and input circuit structure. The input network can be obtained with the help of tools like Synopsys Design Compiler and FlowMap [Cong and Ding 1994]. First, an appropriate initial folding level is computed, based on which RTL modules are partitioned into connected LUT clusters. An area check is performed here in order to verify if the module partition meets the area constraint. Then NanoMap uses FDS [Paulin and Knight 1989] to assign LUTs and LUT clusters to folding stages and balance interfolding stage resource usage, and produces the LUT network of each temporal folding stage.

—*Temporal Clustering* (Steps 8–9). These steps take the flattened LUT network as input and cluster the LUTs into macroblocks (MBs) and SMBs to minimize the need for global interconnect and simplify placement and routing. As opposed to the traditional clustering problem, each hardware resource, that is, LE, MB, or SMB, is temporally shared by logic from different temporal folding stages. Temporal folding necessitates that both intrastage and interstage data dependencies be jointly considered during LUT clustering. Note that the folding stages need not be limited to one plane, that is, temporal clustering can span planes. Verifying if the area constraint is met is done after clustering. If it is met, placement is invoked. Otherwise, NanoMap returns to the logic mapping step.

—*Temporal Placement* (Steps 10–14). These steps perform physical placement and minimize the average length of inter-SMB interconnects. They are implemented on top of an FPGA place-and-route tool, VPR [Betz and Rose 1997], to provide inter-folding stage resource sharing. Placement is performed in two steps. First, a fast placement is used to derive an initial starting point. A low-precision routability and delay analysis is then performed. If the analysis indicates success, a detailed placement is invoked to derive the final placement. Otherwise, several attempts are made to refine the placement and if the analysis still does not indicate success, NanoMap returns to the logic mapping step.

—*Routing* (Step 15). This step uses the VPR router to generate intra-SMB and inter-SMB routing. After routing, the layout for each folding stage is obtained and the configuration bitmap generated for each folding cycle.

—*Complexity of the Algorithm.* In each iteration of the loop, the most computationally intensive steps are FDS and temporal placement. If n is the number of nodes (LUT/LUT cluster) to be scheduled in the design, then the complexity of FDS is $O(n^3)$ [Paulin and Knight 1989]. If r is the number of SMBs after clustering, the complexity of placement is $O(r^{4/3})$ [Marquardt et al. 2000]. Since $n = Cr$, where C is a constant, the total complexity of FDS and placement is still $O(n^3)$. We will see later that the maximum number of iterations required is related to the maximum logic depth of the circuit. If the logic depth is s, the complexity of NanoMap is $O(sn^3)$. If in the FDS step, sorting

is used first, the run time of FDS can be reduced to $O(n^2)$. Hence, the total complexity of NanoMap in this case is reduced to $O(sn^2)$. Note that plane separation and circuit parameter search are performed only once before the iterations start and their complexity is $O(Bn)$, where B denotes the number of inputs of the circuit.

In the following sections, we describe these steps in detail.

4. LOGIC MAPPING

In this section, we discuss the steps involved in logic mapping.

4.1 Plane Separation

Plane separation is the key part of circuit parameter search, which collects the necessary circuit parameters for choosing an appropriate folding level. In this step, we first need to analyze the input design structure, such as how many planes are contained in the design, the logic depth in each plane, plane width, which is the maximum number of LUTs at any logic depth in the plane, etc. First, the circuit structure analysis involves identifying each plane and labeling each module/LUT belonging to it. Then each plane is searched and the above-mentioned parameters obtained.

To identify each plane, a depth-first search algorithm is used. The algorithm begins the search by enumerating every input to the circuit as the root node, and assigning it plane index 1. Then from each root node, the algorithm explores the structure, as far as possible, along each branch and assigns the node (module or LUT) along its path the current plane index. The plane index is increased by one when the algorithm reaches plane registers. If a node is on multiple paths with different plane indices, the least index is used since it guarantees the correct execution sequence and produces a minimal number of planes, that is, results in a smaller delay. Figure 3 gives an example illustrating plane separation. The number inside the parentheses indicates the plane index of the node. We can see that for the multiplexer (MUX) selected by $m2$ on the path from *PI port1*, the plane index is 1. However, on the path from *reg*3, its plane index is three. Based on the policy that the least plane index is assigned, this MUX is labeled with plane index 1. The search algorithm is summarized in Algorithm 1. Note that we use path index in the algorithm to differentiate among multiple paths in case there is a loop along one path. We can see that this plane separation algorithm favors delay optimization. To adjust it for area optimization, modules with multiple plane indices can be assigned to the appropriate plane to balance the number of modules or total number of LUTs among the planes.

4.2 Folding Level Computation

After obtaining the circuit parameters, they are used to compute the initial folding level and adjust the folding level gradually later. Choosing an appropriate folding level is critical to achieving the best area-delay trade-off for the specific optimization objective. As discussed earlier, folding level computation depends on the input circuit structure, which is obtained by identifying each

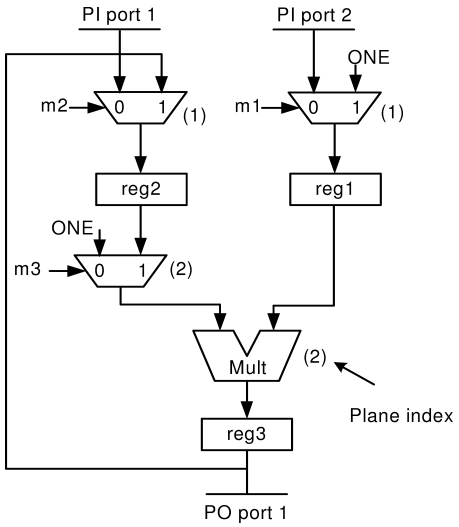

Fig. 3. Plane separation for multiple paths using the least plane index.

Algorithm 1: Algorithm for plane identification

```
1  Plane_search(circuit_graph G)
2  /*If there are multiple plane indices for one node, choose the least */
3  list L = empty;
4  boolean next = false;
5  foreach input e_i of the circuit do
6      set current plane index p = 1;
7      add edge with current plane index (e_i, p) to L;
8      while L is not empty do
9          remove pair (e, p) from the end of L;
10         foreach leaf node n of edge e do
11             if (the plane index p_n of node n is not set) or ((the current path index ≠
               the label of the node) and (p < p_n)) then
12                 set p_n = p;
13                 set next = true;
14                 set the label of node n to be the current path index i;
15             if next = true then
16                 for each output edge e_n of node n do
17                     if node n is a plane register then add pair (e_n, p + 1) to L;
18                     else add pair (e_n, p) to L;
```

plane and obtaining the circuit parameters within each plane. We summarize the necessary circuit parameters below:

—Number of planes in input circuit: num_plane

—Number of LUTs in plane i: num_LUT_i

—Maximum number of LUTs among all the planes:
$LUT\_max = max\{num\_LUT_i\}$, $i=1,...,num\_plane$

—Plane width in plane i: $width\_plane_i$

—Maximum plane width among all the planes:
$width\_max = max\{width\_plane_i\}$, $i=1,...,num\_plane$

—Logic depth of plane i: $depth_i$

—Maximum logic depth among all the planes:
$depth\_max = max\{depth_i\}$, $i=1,...,num\_plane$

—Area constraint, for example, the available number of LEs:
$available\_LE$

—Delay constraint, or sample period, for example, how often new data arrive at the circuit inputs: $sample\_period$

—Number of reconfiguration copies in each nano RAM:
num_reconf

Given the specified optimization objective and constraint, the best folding level is computed using the given parameters. There can be various optimization objectives and constraints. We show how to target the following three design objectives. Similar procedures can target other objectives.

—minimize delay with/without area constraint;

—minimize area with/without timing constraint;

—minimize area-time (AT) product;

4.2.1 *Delay Minimization.* Suppose the optimization objective is to minimize delay. If there is no area constraint, we can first use no-folding and then search around the neighboring folding levels to obtain the shortest delay, as discussed in Part I. If an area constraint is given, it needs to be satisfied first, then the best possible delay obtained. There are two scenarios that need to be considered:

—Multiple planes are allowed to share resources: Such a scenario is possible when there is no feedback across planes. We first perform cross-plane folding in which we stack all the planes together, that is, resources are shared across all planes, since this does not increase circuit delay but reduces area. Then, if LUT_max is larger than $available\_LE$, each plane is folded internally to further reduce area. Since circuit delay is equal to the plane cycle times the number of planes in the circuit, the plane cycle has to be minimized under the area constraint.

Initial Folding Level Selection. Initially, we assume that the logic is balanced well in the plane and compute the minimum number of folding stages required within each plane:

$$\#folding\_stage\_min = \left\lceil \frac{LUT\_max}{available\_LE} \right\rceil. \tag{1}$$

Since the number of folding cycles should be kept the same in each plane, we use the maximum logic depth to compute the maximum initial folding level:

$$initial\_level\_max = \left\lceil \frac{depth\_max}{\#folding\_stage\_min} \right\rceil. \tag{2}$$

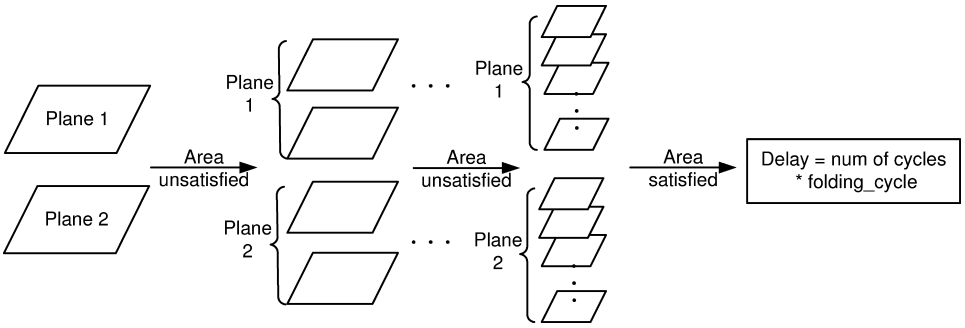

Fig. 4. An example for delay optimization under area constraint, assuming cross-plane resource sharing.

Then, since for most circuits, the logic is not well-balanced along the critical path, we try to push to another extreme and obtain a minimum initial folding level using the maximum plane width:

$$initial\_level\_min = \left\lceil \frac{available\_LE}{width\_max} \right\rceil. \tag{3}$$

We also need to find the minimum folding level, *min_level*, allowed by *num_reconf*:

$$min\_level = \left\lceil \frac{depth\_max * num\_plane}{num\_reconf} \right\rceil. \tag{4}$$

Hence, the final minimum initial folding level is:

$$initial\_level\_min = max\{min\_level,\ initial\_level\_min\}. \tag{5}$$

The initial folding level can be chosen between *initial_level_min* and *initial_level_max*. We found through experiments that for RTL mapping, the best folding level is close to *initial_level_min*, while for gate-level mapping, the best folding level is close to *initial_level_max*. Hence, the initial folding level can be selected based on the input circuit or can just be computed as the average of *initial_level_min* and *initial_level_max* for simplicity.

Folding Level Adjustment. Using the chosen folding level, NanoMap performs FDS and temporal clustering to obtain the area required. If the area constraint is not satisfied, the folding level is decreased by one. Otherwise, the folding level can be increased by one and it can be verified if the area constraint is still satisfied and delay reduced. NanoMap iterates until the area constraint is met or the folding level reduces to the minimum allowed, that is, *min_level*. Figure 4 illustrates one of the optimization approaches.

—Multiple planes are not allowed to share resources: Such a scenario is possible if the RTL circuit is pipelined and, hence, the different pipeline stages need to be resident in the FPGA simultaneously. In this scenario, temporal logic folding can only be performed within each plane. Then *LUT_max* or *width_max* are replaced with $\sum_i num\_LUT_i$ or $\sum_i width\_plane_i$ in the folding level computation.

4.2.2 *Area Minimization.* If the objective is to minimize the mapping area without timing constraint, then as much logic folding as possible should be performed. Hence, inside the plane, the minimal folding level, for example, folding level-1 or one bounded by *num_reconf*, should be used. If a timing constraint is specified, for example, a sample period constraint, denoting the allowed time between successive input samples, NanoMap ensures that the pipeline stage delay of the mapped circuit is less than *sample_period* to guarantee the required throughput. It then obtains the best area. We again consider two scenarios:

—Multiple planes are allowed to share resources: We first stack the planes and try to use the minimum folding level in each plane and then gradually increase it to meet the timing constraint, since the sample period cannot be used to compute the initial folding level directly.

Initial Folding Level Selection. It includes two parts: number of planes stacked together for one pipeline stage and the folding level used inside the plane. As discussed above, we first try the minimum folding level inside the plane. We can see that if *num_reconf* is larger than *depth_max*, level-1 folding is allowed in each plane. Then the number of planes contained in each pipeline stage, *num_folds*, can be obtained as:

$$num\_folds = \left\lfloor \frac{num\_reconf}{depth\_max} \right\rfloor \tag{6}$$

Otherwise, if *num_reconf* is less than or equal to $depth\_max$, one plane is contained in one pipeline stage ($num\_folds = 1$). The minimal folding level inside the plane is set by $depth\_max$ divided by *num_reconf*.

Folding Level Adjustment. The delay for one pipeline stage is equal to $num\_folds \times plane\_cycle$. If the stage delay is larger than *sample_period*, *num_folds* has to be decreased first, then the folding level has to be increased to reduce the pipeline stage delay until the timing constraint is satisfied. Then we can obtain the smallest possible area under the timing constraint. Figure 5 illustrates the discussed procedure.

—Multiple planes are not allowed to share resources: Logic folding is only performed within planes, as discussed before. The NanoMap iteration starts from the minimum folding level and the folding level is increased by one each time, until the timing constraint is met.

4.2.3 *AT Product Minimization.* To minimize the AT product, either area or delay or both need to be minimized. Through our experiments on benchmarks, we found that the delay just increases a little when the folding level is decreased by one, while the area reduces significantly. Hence, to minimize the AT product, the key is to reduce the area significantly. The same procedure as the one for area optimization can be followed. However, since in folding level-1, every temporary result has to be stored in a flip-flop, this results in an area increase. Thus, sometimes, the area for folding level-1 may be close to the area for folding level-2. Hence, to account for this uncertainty, the adjacent folding levels to the chosen folding level are also explored and the best result selected.

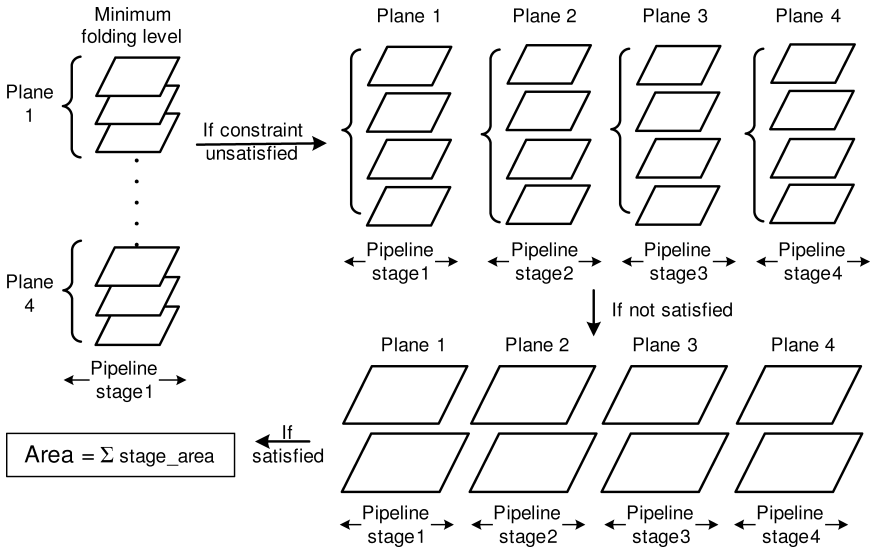

Fig. 5. An illustration for area optimization under timing constraint assuming cross-plane resource sharing.

4.3 Module Library

If the input design is at the RTL, the RTL modules need to be expanded to the LUT level. This expansion can be obtained from the module library, which may be manually designed, targeting area or delay optimization, or automatically generated for regular architectures. The module implementations for required folding levels are provided in the library. Figure 6 shows a part of the library for level-1 folding. It can be seen that for each module, the library provides information on the module name, module function and type (for example, adder can be ripple-carry or carry-select or another type), its bit-width, logic depth, input/output, number of LUT clusters, realization of each LUT cluster, and the number of LUTs used. For each LUT contained in the LUT cluster, the labels inside the parentheses specify the m inputs of the LUT ($m = 4$ in the example), followed by its output, separated by a semicolon. The 16-bit binary number indicates the SRAM configuration for the LUT, which is precomputed to realize the module function. *reg num* gives the number of flip-flops needed to store the temporary computation results from the LUT cluster. This information is used in the LUT scheduling step later.

4.4 RTL Module Partitioning

As discussed in Section 2, to speed up the mapping process, we can take advantage of the RTL module hierarchy and simply slice it into several connected LUT clusters, with each cluster's logic depth being less than or equal to the chosen folding level. During the partitioning process, if the folding level is p, the LUTs with logic depth more than $(n-1)p$ and no more than np are grouped into the n^{th} cluster. Then each cluster is processed in its entirety in the following scheduling step. At the same time, the inputs/outputs of the module are

```
name add2
operation add
type RC
total_LUT 2
width 2
depth 1
input a[0:1] b[0:1]
output c[0:1]
cluster 1
cluster_input a[0] b[0] a[1] b[1]
cluster_output c[0] c[1]
LUT1 (a[1] a[0] b[1] b[0];c[0]) :  0001001101101100
LUT2 (a[1] a[0] b[1] b[0];c[1]) :  1110110010000000
reg 2
```

Fig. 6. Part of the module library.

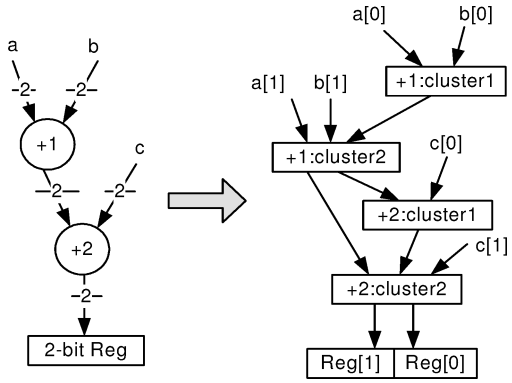

Fig. 7. Illustrating module partitioning.

separated into bit lines that connect various LUT clusters. Based on each selected folding level, the mapper can simply fetch the corresponding cluster-based module implementation from the library (this is introduced later). Figure 7 shows a simple module partitioning example. The partition results in a network of LUTs/(LUT clusters) which is input to the scheduling step. Note that if there is an area constraint, NanoMap first checks to see whether the size of the partitioned LUT cluster exceeds the area constraint. If so, the folding level has to be decreased and partitioning repeated.

4.5 Force-Directed Scheduling

FDS [Paulin and Knight 1989] is a popular scheduling technique in high-level synthesis. It uses an iterative approach to obtain the schedule of operations in order to minimize overall resource usage. Resource usage is modeled as a force. Scheduling of an operation to some time slot, which results in minimum force, indicates a minimum increase in resource usage. Force is calculated based on distribution graphs (DGs), which describe the probability of resource usage for a type of operation in each time slot.

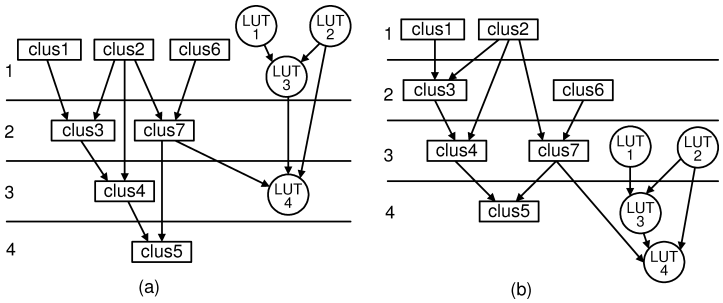

Fig. 8. Schedules of LUTs and LUT clusters in a plane: (a) ASAP schedule, (b) ALAP schedule.

We use FDS in another scenario. We modified it to assign the LUT or LUT cluster to folding stages and balance the resource usage of the folding stages. Based on the underlying NATURE architecture instance, the LE usage in each folding cycle is computed differently.

—Since the number of flip-flops inside each LE is limited, the LE usage in each folding cycle is dependent on both the LUT computations and register storage operations performed in parallel. Thus, two DGs, one describing the resource usage of LUT computations and another for register storage usage, have to be built.

—When the number of flip-flops inside each LE is adequate for storing all temporal results for all folding stages, then LE usage is only determined by LUT computations in each folding cycle. For this case, only the LUT computation DG is needed.

Next, we first discuss how to create the two DGs, and then how the forces are calculated based on the two DGs.

4.5.1 *Creation of LUT Computation DG.* A LUT computation DG models the aggregated probability distribution of the potential concurrency of LUT/(LUT cluster) computations within each folding cycle. To compute the probability distribution of each LUT or LUT cluster, their time frames, $time\_frame_i$, or feasible time interval, are first obtained. Time frames are defined as the span from the folding cycle it is assigned to in the as-soon-as-possible (ASAP) schedule to the folding cycle it is assigned to in the as-late-as-possible (ALAP) schedule. To illustrate this, we present an example in Figure 8. As shown in the figure, since the folding level can be larger than one, we allow the chaining of LUTs in order to provide more flexibility to the mapper. We can see that $time\_frame_{LUT2}$ spans folding cycles 1 to 3, denoted as [1, 3]. Here, clus$i$ denotes LUT cluster i. Figure 9 illustrates the *time_frame* of each LUT and LUT cluster, as shown in the example of Figure 8, at their initial status before scheduling.

If a uniform probability distribution is assumed, the probability that a computation i is assigned to a feasible folding cycle j within its time frame equals $1/|time\_frame_i|$ for $j \in time\_frame_i$ as the number on top of the bars shown in Figure 9 indicates. Then the LUT computation DG value in folding cycle j,

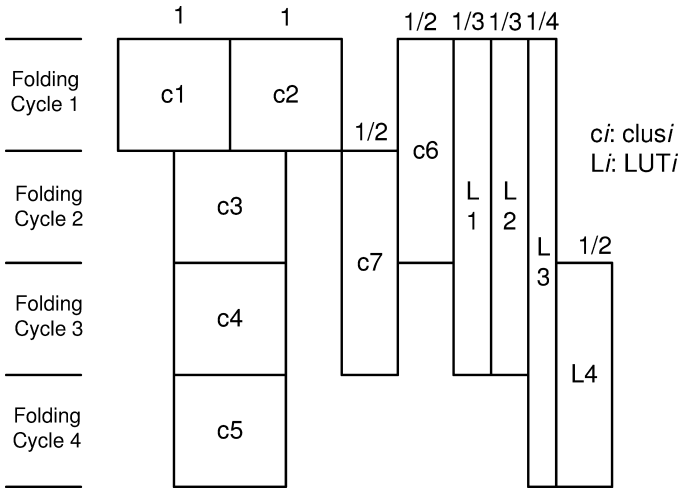

Fig. 9. Time frames of the LUT/(LUT cluster) computations (initial status).

$LUT\_DG(j)$, is the sum of the probabilities of all the computations assigned to this folding cycle:

$$LUT\_DG(j) = \sum_{i=1}^{N} \frac{1}{|time\_frame_i|} * weight_i, \tag{7}$$

where $weight_i$ is 1 for a LUT or equal to the number of LUTs in a LUT cluster.

4.5.2 *Creation of Register Storage DG.* Register storage DG models the distribution of register storage usage. We adopt a similar procedure as above to model the flip-flop usage inside the LE. A storage operation is created at the output of every source computation that transfers a value to one or more destination computations in a later folding cycle. If both the source and destinations of a storage operation are scheduled, the distribution of the storage operation equals its *lifetime*, which begins from the folding cycle of the source and ends at the folding cycle of the last destination. We assume that the results are stored at the beginning of each folding cycle. If one or more of the source or destinations are not scheduled, we have to obtain a probabilistic distribution. If the storage must exist in some cycle, its distribution probability will be 1 in that cycle.

The storage operations in one part of the example in Figure 8 are shown in Figure 10. The following heuristic is used to quickly estimate the resulting storage distribution. First, *ASAP_life* and *ALAP_life* of a storage operation are defined as its lifetime in the ASAP and ALAP schedules, respectively. For example, in Figure 10, the output of source computation LUT2, that is, storage **S**, transfers the value to destination computations LUT3 and LUT4. In the ASAP schedule, **S** begins at folding cycle 2 and ends at folding cycle 3. Hence, $ASAP\_life_{\mathbf{S}} = [2, 3]$ and the length of *ASAP_life*: $|ASAP\_life_{\mathbf{S}}| = 2$. Similarly, in the ALAP schedule, **S** begins at folding cycle 4 and ends at folding cycle 4, which results in $|ALAP\_life_{\mathbf{S}}| = 1$.

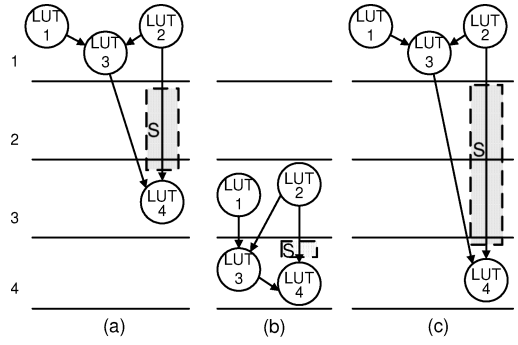

Fig. 10. Storage lifetimes for all schedules: (a) ASAP lifetime, (b) ALAP lifetime, and (c) maximum lifetime.

The longest possible lifetime *max_life* for the storage operation is the union of its *ASAP_life* and *ALAP_life*, whose length is obtained as:

$$|max\_life| = (ALAP\_life\_end - ASAP\_life\_begin + 1). \tag{8}$$

For the example, **S** begins in folding cycle 2 in the ASAP schedule, that is, $ASAP\_life\_begin_{\mathbf{S}} = 2$. Its lifetime ends in cycle 4 in the ALAP schedule, that is, $ALAP\_life\_end_{\mathbf{S}} = 4$. Thus, the length of the maximum lifetime for **S**, for example, $|max\_life_{\mathbf{S}}| = 3$.

If *ASAP_life* overlaps with *ALAP_life*, the overlap time, *overlap*, is the intersection of *ASAP_life* and *ALAP_life*, whose length is similarly obtained as:

$$|overlap| = (ASAP\_life\_end - ALAP\_life\_begin + 1). \tag{9}$$

Within the overlap time, a storage operation must exist with probability 1. For the example, there is no overlap time for **S**. Then an estimate of the average length of all possible lifetimes can be obtained by:

$$avg\_life = \frac{|ASAP\_life| + |ALAP\_life| + |max\_life|}{3}. \tag{10}$$

Next, the probability of a storage operation performed for a LUT or LUT cluster computation i in folding cycle j can be calculated as follows.

When j is outside of $overlap_i$ and $j \in max\_life_i$,

$$storage_i(j) = \frac{avg\_life_i - |overlap_i|}{|max\_life_i| - |overlap_i|} * weight_i. \tag{11}$$

When j is within $overlap_i$, which means a storage operation must be performed, then

$$storage_i(j) = weight_i. \tag{12}$$

The process is carried out for all storage operations, and separate probabilities due to N LUTs and LUT clusters in folding cycle j are added to obtain a single storage DG as follows.

$$storage\_DG(j) = \sum_{i=1}^{N} storage_i(j), \quad j \in max\_life_i. \tag{13}$$

4.5.3 *Calculation of Forces.* The FDS algorithm uses force to model the impact of scheduling of operations on resource usage. It is much like that exerted by a spring that obeys Hooke's law: $F = Kx$, where K is given by the value of $DG(j)$ which presents the probability of resource usage concurrency. A higher force implies higher concurrency of runtime operations, which requires more resources in parallel. The force is contributed by both self-force and the forces of predecessors and successors.

Self-Force. The force in cycle j is calculated based on DGs as:

$$force(j) = DG(j) * x(j), \tag{14}$$

where $DG(j)$ is either $LUT\_DG(j)$ or $storage\_DG(j)$, and $x(j)$ is the increase (or decrease) in the probability of computation in cycle j due to the scheduling of the computation. For example, before scheduling, the computation has a uniform probability of being scheduled in each folding cycle in its time frame. If in a scheduling attempt, the computation is scheduled in folding cycle a, the probability of the computation being scheduled in folding cycle a will increase to 1 and the probability of it being scheduled in other folding cycles will decrease to 0. The self-force associated with the assignment of a computation i, whose time frame spans folding cycles a to b, to folding cycle j is defined as the sum of all the resulting forces in its time frame:

$$\begin{aligned} self\_force_i(j) &= DG(j) * x(j) + \sum_{k=a,k\neq j}^{b} [DG(k) * x(k)] \\ x(j) &= (|time\_frame_i| - 1)/|time\_frame_i| \\ x(k) &= -1/|time\_frame_i|. \end{aligned} \tag{15}$$

Resource usage can be dictated by either LUT computations or storage operations. Assuming there are h LUTs and l flip-flops in one LE, the self-force for scheduling a LUT or LUT cluster i in folding cycle j is determined by both the self-force from the LUT computation and self-force from storage operations as

$$max\{LUT\_self\_force_i(j)/h, storage\_self\_force_i(j)/l\}, \tag{16}$$

when $LUT\_self\_force_i(j)$ and $storage\_self\_force_i(j)$ are positive. $LUT\_self\_force_i(j)$ and $storage\_self\_force_i(j)$ are computed using Equation (15), based on the LUT computation and storage DGs. If the forces are negative, the equation is adjusted to:

$$min\{LUT\_self\_force_i(j)/h, storage\_self\_force_i(j)/l\}. \tag{17}$$

Note that it can never be the case that the LUT self-force is negative and the corresponding storage self-force is positive or vice versa.

To illustrate the procedure, let us schedule clus6 (see Figure 8) as an example. If it is scheduled in cycle 1, the probability of the cluster will change from 1/2 to 1 in cycle 1 and from 1/2 to 0 in cycle 2. Assuming there are four LUT computations and one output in clus6, the resulting *self_force* in cycle 1 is $self\_force(1)$ due to $x(1)$:

$$LUT\_DG(1) * weight * x(1) = 5.92 * 4 * 0.5 = 11.84$$
$$storage\_DG(1) * weight * x(1) = 0 * 1 * 0.5 = 0.$$

According to Equation (16), *self_force*(1) is 11.84. *self_force*(1) due to $x(2)$:

$$LUT\_DG(2) * weight * x(2) = 4.92 * 4 * (-0.5) = -9.84$$
$$storage\_DG(2) * weight * x(2) = 4.88 * 1 * (-0.5) = -2.44.$$

Therefore, we get the self-force due to scheduling of clus6 in cycle 1 as $11.84 - 9.84 = 2$.

Similarly, we can compute the self-force from assigning clus6 to cycle 2. *self_force*(2) is then $9.84 - 11.84 = -2$.

Comparing the self-forces under the two assignments, we can see that scheduling clus6 in cycle 1 results in more force than in cycle 2. Since less the force is, less the concurrency the assignment will result in, the computation should be scheduled into cycle 2.

Predecessor and Successor Forces. The second part of the force is the predecessor and successor forces. Assigning a LUT computation to a specific folding cycle often affects the time frame of its predecessors and successors, which in turn creates additional forces affecting the original move. Equation (15) is used to compute the force exerted by each predecessor or successor. The overall force is then the sum of the self-force and forces of predecessors and successors. Then the total forces under each schedule for a computation are compared and the computation is scheduled into the folding cycle with the lowest force.

For example, if clus6 was tentatively assigned to cycle 2, the succeeding clus7 would implicitly be assigned to cycle 3 and LUT4 to cycle 4. The time frames of the other nodes would not be affected. Assuming four LUTs and one output in clus7 too, the successor force from implicitly assigning clus7 to cycle 3 can be obtained as:

$$LUT\_DG(3) * weight * x(3) = 4.92 * 4 * 0.5 = 9.84$$
$$storage\_DG(3) * weight * x(3) = 4.55 * 1 * 0.5 = 2.28$$
$$LUT\_DG(2) * weight * x(2) = 4.92 * 4 * (-0.5) = -9.84$$
$$storage\_DG(2) * weight * x(2) = 4.88 * 1 * (-0.5) = -2.44.$$

Hence, the successor force due to clus7 is $9.84 - 9.84 = 0$. Similarly, we can obtain the successor force due to LUT4 as:

$$LUT\_DG(4) * weight * x(4) = 2.75 * 1 * 0.5 = 1.38$$
$$storage\_DG(4) * weight * x(4) = 3.67 * 1 * 0.5 = 1.84$$
$$LUT\_DG(3) * weight * x(3) = 4.92 * 1 * (-0.5) = -2.46$$
$$storage\_DG(3) * weight * x(3) = 4.55 * 1 * (-0.5) = -2.28.$$

Thus, the successor force from assigning LUT4 is $1.84 - 2.46 = -0.62$. Hence, the total successor force is -0.62. Then considering all the related forces, clus6 should be scheduled to cycle 2 to give the least concurrency.

Algorithm 2: Force-directed scheduling

```
while there are LUT/(LUT cluster) computations to be scheduled do
    evaluate its time frame using ASAP and ALAP scheduling;
    create the LUT computation and register storage DGs;
    foreach unscheduled LUT/(LUT cluster) computation i do
        foreach each feasible clock cycle j it can be assigned to do
            calculate the self-force of assigning node i to cycle j;
            add predecessor and successor forces to self-force to get the total force for
            node i in cycle j;
        select the cycle with the lowest total force for node i;
    Pick the node with the lowest total force and schedule it in the selected cycle;
```

4.5.4 *Summary of the FDS Algorithm.* FDS uses an iterative approach to schedule one computation in each iteration. In each iteration, the LUT computation and register storage DGs are obtained. The LUT or LUT cluster with the minimum force is chosen, and assigned to the folding cycle with the minimum force, which potentially leads to a minimal increase in resource usage. This process continues until all the LUT or LUT cluster computations are scheduled. The pseudo-code of the proposed FDS technique is shown in Algorithm 2. We call this algorithm Heuristic 1. To improve the run time, another approach is to sort the LUT or LUT cluster according to its weight (number of LUTs) first. The node with the largest weight is scheduled first. Since smaller the weight of the node, smaller its impact, our experiments show that the scheduling result is degraded only slightly, but the run time is improved significantly. We name this approach Heuristic 2.

5. TEMPORAL CLUSTERING

During scheduling, nodes of LUTs or LUT clusters are assigned to each folding stage. In the temporal clustering step, for each folding stage, we use a constructive algorithm to assign the network of LUTs to LEs and pack LEs into MBs and SMBs. To construct each SMB, we first choose a LUT cluster with a maximal number of inputs and choose a LUT, which uses a maximal number of its inputs, within that cluster as an initial seed. Then, new LUTs with high attractions to the seed LUT are chosen and assigned to the SMB, until the SMB is fully packed. Then a new LUT seed is selected. The attraction between a LUT i and the seed LUT, $Attraction_{i,seed}$, depends on timing criticality and input pin sharing [Marquardt et al. 1999], as follows:

$$Attraction_{i,seed} = \alpha * Criticality_i + (1 - \alpha) * \frac{Nets_{i,seed}}{G}. \tag{18}$$

The first term on the right hand side models the timing criticality and the second term models the net congestion. In the first term, the timing criticality of LUT i, $Criticality_i$, can be further expanded as:

$$Criticality_i = max_j\{Connection\_Criticality(j)\} + \varepsilon * total\_paths\_affected_i, \tag{19}$$

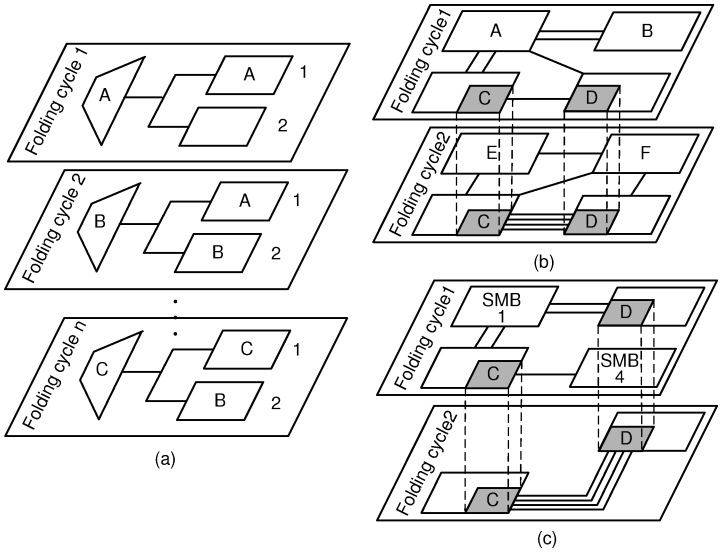

Fig. 11. Illustration of (a) flip-flop usage in temporal clustering, (b) attraction computation in temporal clustering, and (c) placement with logic folding.

$j \in \{\text{inputs of LUT } i \text{ connected to } seed\}$, and

$$Connection\_Criticality(j) = 1 - \frac{slack(j)}{MaxSlack}.$$

Here, *slack* is defined as the amount of delay between the actual arrival time and the required arrival time, and *MaxSlack* is the maximum *slack*. The second part of Equation (19) is used for tie-breaking purposes when multiple nodes have the same criticality. The tie is broken in a manner to most effectively reduce the number of nodes remaining on the critical paths. $total\_paths\_affected_i$ tracks how many critical paths the node is on and ε is a parameter with a small value that ensures the *total_paths_affected* value acts only as a tie-breaker (ε =0.01 worked well in our experiments).

In the second term, $Nets_{i,seed}$ is the number of shared inputs/outputs between the two LUTs, and G is the number of inputs/outputs per LE. α is a parameter between 0 and 1 that allows a trade-off between timing criticality and interconnect demand and can be tuned to the specific optimization. For example, for routing area optimization, we can lay emphasis mainly on the interconnect, and for delay optimization, timing criticality plays the key role.

As opposed to traditional clustering methods, to support temporal logic folding, inter-folding stage resource sharing needs to be considered during clustering. Since due to logic folding, several folding stages may be mapped to a set of LEs, some of the LEs may be used to store the internal results and transfer them to another folding cycle. Hence, there are LEs spanning several cycles and connecting with other LEs in different cycles. Figure 11(a) illustrates an LE with one LUT and two flip-flops. In folding cycle 1, the LUT implements function *A*

Algorithm 3: Temporal clustering

```
1  foreach folding stage i do
2      while there are unclustered LUTs in the folding stage i do
3          found = true;
4          if current SMB curr_SMB not full then
5              if there are LUTs which can be accommodated in curr_SMB then
6                  found LUT_a with the highest attraction with curr_SMB;
7                  select_LUT = LUT_a;
8              else
9                  found = false
10         else
11             found = false
12         if found == false then
13             select the seed LUT LUT_b;
14             select_LUT = LUT_b;
15             create a new SMB and assign it to curr_SMB;
16         assign select_LUT to the available LE in curr_SMB;
17         mark select_LUT as clustered;
18         record LUT mapping and flip-flop usage;
19         reduce the number of available LEs in curr_SMB by one;
```

and stores the computation result in the first flip-flop. The result is maintained until folding cycle n. In the second cycle, the LUT implements function B and if the result also needs to be stored, it has to be stored in flip-flop 2 since flip-flop 1 is already occupied. Suppose in some cycle between the second and n^{th}, all the flip-flops inside the LE are occupied and only the LUT is available. If it is a LUT computation requiring result storage, a new LE is needed to accommodate the LUT computation, since separating the computation and result storage will worsen the delay. Otherwise, if the objective is to optimize area, the LUT computation and its result can be separately mapped to a LUT and flip-flop in different currently available LEs in order to reduce total LE usage. Hence, in temporal clustering, both the LUT mapping and the lifetime of temporal result storage need to be carefully tracked. Moreover, since two LEs (e.g., C and D in Figure 11(b)) may have multiple attractions between them across several folding cycles, when performing temporal clustering, the attractions between them should be set to the maximum attraction over all the cycles.

The pseudo-code for temporal clustering algorithm is shown in Algorithm 3.

6. TEMPORAL PLACEMENT AND ROUTING

We modified VPR to perform placement and routing. Placement uses a two-step simulated annealing approach. It starts with a fast low-precision placement, which has 10X speed-up and only 5% performance degradation. VPR's routability delay analysis is then used to evaluate the quality of this initial placement, which determines whether a high-precision placement or another round of optimization using adjusted logic folding should be invoked.

VPR uses simulated annealing for placement. The functional form of its cost function targeted at net congestion is

$$Cost = \sum_{n=1}^{N_{nets}} \frac{q(n)}{C_{av}} [bb_x(n) + bb_y(n)], \tag{20}$$

where the summation is over all the nets in the circuit. For each net n, $bb_x(n)$ and $bb_y(n)$ denote the horizontal and vertical spans of its bounding box, respectively. $q(n)$ is the pin-count net-weight, which is 1 for nets with three or fewer terminals, and slowly increases to 2.79 for nets with 50 terminals. C_{av} is the average channel capacity (in terms of the number of tracks). Since in NATURE, all channels have the same capacity, C_{av} is a constant. Hence, the congestion cost function is equivalent to a bounding box cost function. We modified the VPR placer to support temporal logic folding, which introduces interfolding stage dependencies. Consider the example in Figure 11(c). In folding cycle 1, since there are few connections between C and D, they may be placed far apart. However, such a placement would not be good for folding cycle 2 in which C and D communicate a lot. Hence, the pin-count for such a pair of SMBs should consider the connections in all the folding stages. When swapping such pairs, they should be swapped in all the folding stages where they appear.

Recently, timing-driven placement has also been added to VPR, and we have similarly modified it to support logic folding.

After the placement is done, detailed routing can be performed using VPR to connect all the SMBs in each folding stage and finish the mapping process. We added the ability to tackle direct links in the router in order to support the interconnect structure in NATURE. Routing in VPR can be conducted in a timing-driven fashion. Hence, the router tends to first use direct links, then length-1 and length-4 wire segments and finally global interconnects (these are the four types of interconnects available in NATURE). Note that a length-i interconnect spans i SMBs. Finally, based on the placement and routing result in each folding cycle, the reconfiguration bits for each LE/switch can be generated.

7. EXPERIMENTAL RESULTS

In this section, to show the advantages of NATURE and efficiency of NanoMap, we present experimental results for the mapping of twelve RTL/gate-level benchmarks to typical instances of NATURE using NanoMap. NATURE is a family of architectures, which may vary in the number of inputs (m) and flip-flops (l) in an LE, number of LEs (n_1) in an MB, number of MBs (n_2) in an SMB, and even the type of on-chip nano RAM. We first present the mapping results based on the baseline design using NRAM, then discuss the effects of architectural variations on the mapping results.

7.1 Experimental Setup

Based on the discussion in Part I, we use an architectural instance with one four-input LUT, two flip-flops in an LE, four LEs in an MB and four MBs in an SMB as the baseline design to obtain good area-delay trade-offs. All experiments

are based on a 70nm technology (i.e., both the CMOS logic and NRAM are implemented in this technology). To fully explore the potential of logic folding, we assume that a varying number of reconfiguration sets, k, is available in NRAMs, depending on the application. We also show the trade-offs when the size of NRAM is fixed to 16 sets instead, which means the circuit can be folded at most 16 times (for such a case, the NRAMs only occupy 10% of chip area). Among the twelve benchmarks we targeted, GCD is Euclid's greatest common divisor[1] implemented in Muttreja et al. [2008]. ex1 is the circuit shown in Figure 1, but with a bit-width of 16. ex2 is an RTL circuit from Lingappan et al. [2006]. Paulin and diffeq are two differential-equation solvers implemented in Lingappan et al. [2006] and Zhong and Jha [2005], and FIR and Biquad are two types of digital filters. ASPP4 is an application-specific programmable processor synthesized in Ghosh et al. [1999]. des is a combinational benchmark from Lisanke [1988]. wavelet and DCT are two mathematical functions implementing wavelet transform and discrete cosine transform computations, respectively. Finally, b20 is a gate-level benchmark from the ITC'99 benchmark suite.

7.2 Experimental Results for AT Product Optimization

We first map all benchmarks under the AT product minimization objective to show the logic density benefits of temporal logic folding relative to the traditional no-folding case. The corresponding area (no. of LEs is used as a proxy for area because of the regular architecture), delay and AT product improvement with respect to the no-folding case for the scenarios without and with limitations on k are shown in Table I. The first five columns describe the benchmark name and structure. Mapping results for no-folding, k enough and $k = 16$ cases are shown in second and third part of the table, respectively. We can see that AT product optimization is achieved with folding level-2 in most cases when there is no restriction on k, because an increase in circuit delay is more than overcome by the dramatic reduction in area. However, for level-1 folding, since more temporary results need to be stored in flip-flops, and the number of extra LEs required to accommodate these extra flip-flops may in some cases outweigh the reduction in LEs required for the combinational logic, level-1 folding is not always better than level-2 folding for AT product optimization. Here, *num_folds* indicates the number of planes cross-plane folding is performed on, as discussed in Section 4.2. The average reduction in the number of LEs is $38.1X$ ($6.5X$) and in the AT product $7.8X$ ($4.6X$) at the expense of a $2.9X$ ($1.6X$) increase in delay for large enough k (k limited to 16). The delay increase mainly comes from having to choose the clock cycle based on the longest folding cycle. Hence, it is crucial to make the folding stages well-balanced.

NanoMap can target many different optimization objectives. Typical objectives are: (i) minimization of circuit delay with or without an area constraint, (ii) minimization of area with or without a delay constraint, (iii) minimization of the AT product, and (iv) finding a feasible implementation under both area and delay constraints. To save space, we choose different optimization objectives for different benchmarks and present the results in Table II. In Column 2, we

[1]High level synthesis benchmarks. http://www.ece.vt.edu/mhsiao/hlsyn.html.

Table I. Circuit Mapping Results for AT Product Optimization

Circuit	#Planes	Max plane depth	#LUTs	#Flip flops	No-folding	
					#LEs	Delay (*ns*)
GCD	2	19	143	34	175	24.46
ex1	2	25	587	80	667	39.86
ex2	3	26	627	114	739	64.50
diffeq	2	32	837	96	933	49.70
Biquad	2	28	1207	64	1271	23.55
Paulin	2	28	1290	114	1402	67.80
ASPP4	2	27	2002	114	2114	62.20
des	1	11	2166	0	2166	14.18
wavelet	4	26	6808	960	7768	101.64
DCT	5	26	6866	930	7796	113.25
b20	3	42	7828	429	7828	153.69
FIR	20	33	10989	320	10989	64.68
Circuit	AT prod. optimization (k enough)					
	num_folds	Fold. level	#Fold. cycles	#LEs	Delay (*ns*)	AT prod. improv.
GCD	2	4	10	34	30.70	4.10X
ex1	2	2	26	80	59.68	5.57X
ex2	3	2	39	84	88.92	6.38X
diffeq	2	4	16	128	68.80	5.27X
Biquad	1	2	14	136	40.18	5.48X
Paulin	2	2	28	128	82.88	8.96X
ASPP4	2	2	28	200	87.64	7.50X
des	1	2	6	390	19.02	4.14X
wavelet	4	2	52	373	217.36	9.74X
DCT	5	2	65	336	264.55	9.93X
b20	3	4	33	697	270.66	6.38X
FIR	20	1	660	32	1094.80	20.29X
Circuit	AT prod. optimization ($k = 16$)					
	num_folds	Fold. level	#Fold. cycles	#LEs	Delay (*ns*)	AT prod. improv.
GCD	2	4	10	34	30.70	4.10X
ex1	2	4	14	139	51.38	3.72X
ex2	3	6	15	160	77.25	3.86X
diffeq	2	4	16	128	68.80	5.27X
Biquad	1	2	14	136	40.18	5.48X
Paulin	2	4	14	240	70.84	5.60X
ASPP4	2	4	14	312	69.44	6.07X
des	1	2	6	390	19.02	4.14X
wavelet	4	8	16	1062	158.08	4.70X
DCT	1	2	13	800	278.85	3.97X
b20	1	4	11	886	279.18	4.87X
FIR	1	5	7	1355	157.92	3.32X

mention the optimization objective. In Columns 3 and 4, we list the constraints (area or delay or neither). These results show the versatility of NATURE and NanoMap. A significant side-benefit of the dramatic area reduction made possible by logic folding is the associated reduction in the need for a deep interconnection hierarchy in NATURE. Since cycle-by-cycle reconfiguration makes LE utilization very high, the need for global communication greatly reduces. We found that global interconnect usage went down by more than 50% when using

Table II. Circuit Mapping Results for Typical Optimizations

Circuit	Opt. obj.	Area const. (#LEs)	Delay const. (*ns*)	num_folds	Folding level	#LEs	Delay (*ns*)
GCD	Area	–	–	2	2	34	44.08
ex1	Delay	–	–	1	no	667	39.86
ex2	Area	–	85	3	4	144	83.79
diffeq	Area	–	70	2	4	128	68.80
Biquad	Delay	110	–	1	1	102	55.16
Paulin	–	250	80	2	4	240	70.84
ASPP4	Area	–	70	2	4	312	69.44
des	Area	–	–	1	1	302	31.46
wavelet	Delay	–	–	4	no	7768	101.64
DCT	–	600	200	5	4	512	184.10
b20	Delay	–	–	3	no	7828	153.69
FIR	Delay	60	–	20	2	60	752.40

Table III. Circuit Mapping Results from RTL and Gate-Level Descriptions

Circuit	num_ folds	Folding level	RTL		Gate-level		#LEs red. (%)	Delay red. (%)
			#LEs	Delay (*ns*)	#LEs	Delay (*ns*)		
GCD	2	4	34	30.70	34	35.20	0	−14.66
ex1	2	4	139	51.38	144	54.95	−3.60	6.94
ex2	3	4	144	83.79	112	86.52	22.22	−3.26
diffeq	2	4	144	76.50	128	68.80	11.11	10.07
Biquad	1	4	256	30.96	240	30.38	6.25	1.87
Paulin	2	4	256	74.24	240	70.84	6.25	4.58
ASPP4	2	4	383	72.16	312	69.44	18.54	3.77
wavelet	4	4	736	157.08	601	172.48	18.34	−9.80
DCT	5	4	512	184.10	429	212.80	16.21	−15.59
FIR	20	4	120	734.00	92	754.2	23.33	−2.75

level-1 folding as opposed to no-folding. This points to trading interconnect area for increased NRAM area as an attractive alternative for NATURE.

7.3 Comparison of RTL and Gate-Level Mapping

Since NanoMap can start with an input design specified at the RTL or gate-level, there are also two ways to implement a design, thus providing flexibility to designers. For RTL descriptions, since the modules are partitioned into LUT clusters and mapped as one entity, there is less scheduling freedom for each LUT. However, the manually-designed library enables optimization inside LUT clusters. Hence, this may result in some variation in circuit delay when comparing the mappings from a gate-level description and from an RTL description. The average reduction in the number of LEs required using gate-level mapping increases with a folding level increase due to its scheduling freedom advantage. However, the use of LUT clusters in RTL mapping speeds up the mapping process. Table III shows comparisons between mapping results from RTL and gate-level implementations. Comparisons for des and b20 are not reported because we only had their gate-level implementations available to us. The average reduction in the number of LEs required using gate-level mapping is 11.9%, while RTL mapping gives 3.3% delay improvement compared to gate-level mapping. NanoMap was run on a 2*GHz* PC with 1*GB* DRAM under RedHat Linux 9. RTL

Table IV. Experimental Results of Benchmark Biquad for Different n_2

n_2	Area	Logic folding (ns)			No-folding (ns)	AT prod. improv.
		Level-1	Level-2	Level-4		
2	0.8	52.08	38.08	32.55	24.26	5.65X
4	1	54.04	40.18	30.38	23.55	5.48X
6	1.13	55.16	41.44	32.41	23.20	4.97X

Table V. Experimental Results for Different k

k	Area overhead %	AT prod. improv.
16	10.6	4.80X
32	19.2	6.04X
64	27.7	6.83X

mapping is on an average 1.5X faster than gate-level mapping using Heuristic 2 and 30X faster using Heuristic 1. Comparing the two heuristics, Heuristic 1 gives only slightly better results in mapping area and delay than Heuristic 2. However, its run time is more than an order of magnitude longer than that of Heuristic 2 for large circuits. Hence, we only present results for Heuristic 2.

7.4 Experimental Results for Various Architectural Instances

NATURE is a family of architectures. We discuss the effects of varying the values of n_1, n_2, l and k on the mapping results next. Since the number of inputs to an LUT and the types of nano RAM used (their reconfiguration time does not differ much) do not have a large impact on the mapping flow, especially the folding level selection, we will not discuss their variations here.

Variation of n_1 or n_2. Variation of n_1 or n_2 leads to a variation in the number of LEs in an SMB. As discussed in Part I, increasing the number of LEs in an SMB increases the SMB size and, hence, increases the interconnect delay between SMBs, however, enables more communications to be local. Its effects on circuit delay for different folding levels depends on how many communications occur inside SMBs and how many between SMBs. Typically, increasing n_1 or n_2 will favor larger folding levels or even the no-folding case and, hence, favor delay optimization. However, n_1 or n_2 cannot be allowed to become so large that even the delay inside the SMB becomes too large. Since area increases greatly with an increase in the number of LEs contained in an SMB, a large n_1 or n_2 is not the best choice for area optimization, or even AT product improvement. We illustrate the mapping results for the Biquad benchmark in Table IV to show the trend. We keep n_1 unchanged at four and vary n_2. All the results were obtained assuming k to be enough. Since for Biquad, the best folding level for AT product optimization is 2, for the $n_2 = 2$ case, compared to the $n_2 = 4$ case, the delay for level-2 folding is reduced and for no-folding increased. Hence, the AT product improves slightly. For the $n_2 = 6$ case, the situation is the opposite.

Variation of k. As we can see from the mapping results in Table I, $k = 16$ already leads to significant area savings and AT product improvement. However, a larger k can enable even larger benefits of logic folding. In Table I, for the k-enough case, the value of k needed is typically less than 60. Different values of k will obviously lead to different optimal folding level choices. In addition, a

Table VI. Experimental Results of Benchmark Biquad for Different l

l	SMB size	Level-1 folding		Level-2 folding		No-folding (ns)		Opt. folding level	AT prod. improv.
		delay (ns)	$\#SMBs$	delay (ns)	$\#SMBs$	delay (ns)	$\#SMBs$		
2	1	54.04	7	40.18	9	23.55	76	2	5.48X
3	1.53	62.44	5	44.24	9	27.21	76	1	6.63X
4	2.07	54.60	5	50.96	9	29.08	76	1	8.10X

higher value of k will enable a larger AT product improvement. In Table V, we list the chip area overhead for different values of k and the average AT product improvement for the benchmarks.

Variation of l. In our experiments, we observed that temporal logic folding greatly reduces the area for implementing logic, so much so that the number of registers in the design becomes the bottleneck for area reduction. Thus, large benchmarks need more than one flip-flop per LE. In Part I, we found, in general, the best value of l to be two. However, for some designs, even more than two flip-flops per LE may be best. Here, we discuss the effects of variation of l on the mapping and choice of the best folding level. As discussed in Part I, an increase in l will increase the SMB size, however, will reduce the number of SMBs required. If the total area used is reduced, the circuit delay may also be improved. A small folding level favors a large l to store the temporal results, while no-folding favors a small l. A large l facilitates area optimization using logic folding, however, lengthens the delay for the no-folding case. For a specific architectural instance with large l, for example, $l = 4$, usually the best folding level for area or AT product optimization is level-1 folding. Table VI shows the mapping results for Biquad for different l values. k was assumed to be enough for these results as well.

7.5 Power Estimation

Finally, the power estimation results based on the baseline design for the different implementations are shown in Table VII in order to provide an idea of the power consumption required when mapping large benchmarks to NATURE. We estimated the power of the benchmarks using the method discussed in Part I and compared their power consumption for different folding levels with the no-folding case. Figure 12 shows the power consumption of benchmarks for different folding levels. We can see that the no-folding scenario typically results in higher logic, interconnect and leakage power than when using logic folding, as discussed in Part I. Here, we did not consider NRAM leakage in the power consumption for the no-folding case, since traditional FPGAs do not contain NRAMs. Level-1, level-2 and level-4 folding, on an average, require 49%, 48%, and 52% of the power consumption, respectively, compared to the no-folding case.

8. CONCLUSIONS

We presented a circuit mapping methodology for a high-performance nano/CMOS hybrid dynamically reconfigurable architecture called NATURE.

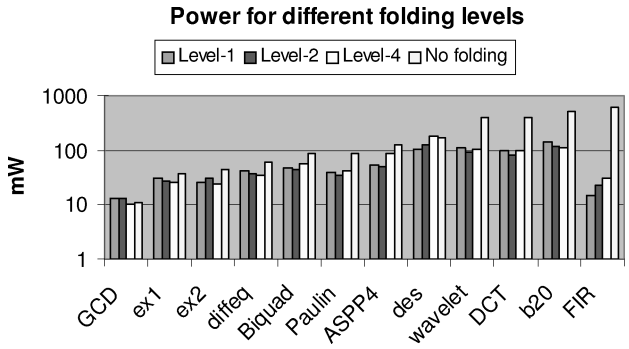

Fig. 12. Power consumption of benchmarks for different folding levels.

Table VII. Power Estimation for Benchmarks

Circuit	Logic folding							
	Folding level	Clock freq. (MHz)	Dynamic power (mW)				Leakage power (mW)	Total power (mW)
			Reconf.	Clock	Logic	Inter.		
GCD	1	621.12	10.50	0.14	0.18	0.57	1.85	13.24
ex1	2	434.78	19.13	0.22	0.37	2.03	4.80	26.55
ex2	4	250.63	14.91	0.13	0.28	2.26	6.59	24.17
diffeq	1	520.83	32.01	0.26	0.44	2.54	6.76	42.01
Biquad	2	348.43	27.60	0.18	0.83	6.31	8.81	43.73
Paulin	2	337.84	21.98	0.17	0.55	4.44	7.23	34.37
ASPP4	4	201.61	48.66	0.28	0.80	10.98	28.64	89.36
des	4	185.53	82.18	0.67	2.95	40.85	55.78	182.43
wavelet	1	349.65	72.19	0.49	1.24	11.47	24.14	109.53
DCT	1	371.75	67.94	0.52	1.17	8.76	21.01	99.40
b20	4	121.92	47.22	0.34	0.99	15.41	47.81	111.77
FIR	2	478.47	16.92	0.11	0.37	1.33	3.85	22.58

Circuit	No-folding						Power red.
	Clock freq. (MHz)	Dynamic power (mW)			Leakage power (mW)	Total power (mW)	
		Clock	Logic	Inter.			
GCD	40.88	0.04	0.14	1.41	9.06	10.65	0.80X
ex1	25.09	0.07	0.33	3.25	32.87	36.52	1.38X
ex2	15.50	0.04	0.23	2.94	41.46	44.67	1.85X
diffeq	20.12	0.07	0.37	5.46	54.10	60.00	1.43X
Biquad	42.46	0.19	1.05	15.16	71.49	87.89	2.01X
Paulin	14.75	0.08	0.40	6.11	80.21	86.80	2.53X
ASPP4	16.08	0.13	0.66	9.93	117.87	128.59	1.44X
des	70.52	0.57	2.93	44.37	125.47	173.34	0.95X
wavelet	9.84	0.27	1.40	15.64	390.10	407.41	3.72X
DCT	8.83	0.24	1.21	13.33	376.87	391.65	3.94X
b20	6.51	0.18	0.99	18.59	476.79	496.55	4.44X
FIR	15.46	0.60	3.13	34.40	557.67	595.80	26.39X

It supports fine-grain runtime reconfiguration and, hence, enables temporal logic folding. Through logic folding, significant logic density improvement and flexibility in performing area-delay trade-offs are achieved.

To fully exploit the features of NATURE, the RTL/gate-level automatic design optimization platform, NanoMap, incorporates temporal logic folding during the logic mapping, temporal clustering and placement steps. It can automatically select the best folding level and use FDS to balance resources across different folding stages. The mapping can be targeted at various optimization objectives and design constraints, providing significant design flexibility.

REFERENCES

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the International Workshop on Field-Programmable Gate Airays*. 213–222.

CONG, J. 1996. Combinational logic synthesis for LUT based field-programmable gate arrays. *ACM Trans. Des. Automat. Electron. Syst. 1*, 145–204.

CONG, J. AND DING, Y. 1994. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table-based FPGA designs. *IEEE Trans. Comput.-Aid. Des. 13*, 1–12.

CUI, Y., ZHONG, Z., WANG, D., WANG, W. U., AND LIEBER, C. M. 2003. High performance silicon nanowire field effect transistors. *Nano Lett. 3*, 149–152.

DEHON, A. 2006. 3D nanowire-based programmable logic. In *Proceedings of the International Conference on Nano-Networks*. 1–5.

GHOSH, I., RAGHUNATHAN, A., AND JHA, N. K. 1999. Hierarchical test generation and design for testability methods for ASPPs and ASIPs. *IEEE Trans. Comput.-Aid. Des. 18*, 357–370.

GOLDSTEIN, S. C. AND BUDIU, M. 2001. Nanofabrics: Spatial computing using molecular nanoelectronics. In *Proceedings of the International Symposium on Computer Architecture*. 178–189.

ITC. 1999. ITC'90 benchmarks. http://www.eerc.utexas.edu/itc99-benchnmarks/bench.html.

ITRS. 2007. International Technology Roadmap for Semiconductors. http://public.itrs.net.

JAVEY, A., GUO, J., FARMER, F. B., WANG, Q., AND WANG, D. 2004. Carbon nanotube field-effect transistors with integrated ohmic contacts and high-k gate dielectrics. *Nano Lett. 4*, 447–450.

LAI, S. 2003. Current status of the phase change memory and its future. In *Proceedings of the International Electronic Devices Meeting*. 10.1.1–10.1.4.

LINGAPPAN, L., RAVI, S., AND JHA, N. K. 2006. Satisfiability-based test generation for nonseparable RTL controller-datapath circuits. *IEEE Trans. Comput.-Aid. Des. 25*, 544–557.

LISANKE, R. 1988. Logic synthesis and optimization benchmarks. Tech. rep., Microelectronics Center of North Carolina.

LUO, Y. 2002. Two-dimensional molecular electronics circuits. *Phys. Chem. 3*, 519–525.

MARQUARDT, A., BETZ, V., AND ROSE, J. 2000. Timing-driven placement for FPGAs. In *Proceedings of the International Symposium on FPGA*. 203–213.

MARQUARDT, A. S., BETZ, V., AND ROSE, J. 1999. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *Proceedings of the International Symposium on FPGAs*. 37–46.

MUTTREJA, A., RAVI, S., AND JHA, N. K. 2008. Variability-tolerant register-transfer level synthesis. In *Proceedings of the International Conference on VLSI Design*. 621–628.

NANTERO. 2008. Nantero. http://www.nantero.com.

PAULIN, P. G. AND KNIGHT, J. P. 1989. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. Comput.-Aid. Des. 8*, 661–679.

RAD, R. M. P. AND TEHRANIPOOR, M. 2006. A new hybrid FPGA with nanoscale cluster and CMOS routing. In *Proceedings of the Design Automation Conference.* 727–730.

SNIDER, G., KUEKES, P., AND WILLIAMS, R. S. 2004. CMOS-like logic in defective, nanoscale crossbars. *Nanotech. 15*, 881–891.

STRUKOV, D. B. AND LIKHAREV, K. K. 2005. CMOL FPGA: A reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices. *Nanotech. 16*, 888–900.

SYNOPSYS. 2009. Synopsys. http://www.synopsys.com.

TEHRANI, S., SLAUGHTER, J. M., DEHERRERA, M., ENGEL, B. N., AND RIZZO, N. D. 2003. Magnetoresistive random access memory using magnetic tunnel junctions. *Proc. IEEE 91*, 703–714.

ZHANG, W. AND JHA, N. K. 2005. ALLCN: An automatic logic-to-layout tool for carbon nanotube based nanotechnology. In *Proceedings of the International Conference on Computer Design*. 281–288.

ZHANG, W., SHANG, L., AND JHA, N. K. 2007. Nanomap: An integrated design optimization flow for a hybrid Nanotube/CMOS dynamically reconfigurable architecture. In *Proceedings of the Design Automation Conference.* 300–305.

ZHONG, L. AND JHA, N. K. 2005. Interconnect-aware low-power high-level synthesis. *IEEE Trans. Comput.-Aid. Des. Integ. Circ. Syst. 24*, 336–351.

Received Augsut 2008; revised January 2009; accepted February 2009 by Ramesh Karri