NanoMap: An Integrated Design Optimization Flow for a Hybrid Nanotube/CMOS Dynamically Reconfigurable Architecture *

Wei Zhang
Dept. of Electrical Engineering
Princeton University,
Princeton, NJ 08544
weiz@princeton.edu

Li Shang
Dept. of Electrical Engineering
Queen's University, Kingston,
ON K7L 3N6
li.shang@queensu.ca

Niraj K.Jha
Dept. of Electrical Engineering
Princeton University,
Princeton, NJ 08544
jha@princeton.edu

ABSTRACT

NATURE is a recently developed hybrid nano/CMOS reconfigurable architecture. It consists of complementary metal-oxide semiconductor (CMOS) reconfigurable logic and interconnect fabric, and carbon nanotube-based non-volatile on-chip configuration memory. Compared to existing CMOS-based field-programmable gate arrays (FPGAs), NATURE increases logic density by more than an order of magnitude and offers cycle-by-cycle run-time reconfiguration capability. As opposed to some other recently proposed hybrid nano/CMOS designs, which mostly rely on the not-yet-mature self-assembly fabrication process, NATURE is compatible with mainstream photolithography fabrication techniques. Thus, NATURE offers a commercially feasible technology with high performance, superior integration density, and excellent run-time flexibility.

In this paper, we present an integrated design and optimization platform 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, placement, and routing. NATURE offers a highly-efficient computation model, called temporal logic folding. A logic circuit can be arbitrarily folded into a sequence of logic stages, which temporally share and execute on the same hardware resource using fine-grain run-time reconfiguration. To effectively leverage this feature, we propose and develop novel mapping techniques which can automatically explore and identify the best temporal logic folding configuration, targeting area, delay or area-delay product. It uses a force-directed scheduling technique to optimize and balance resource usage across different folding cycles. It provides significant design flexibility in performing area-delay tradeoffs under various user-specified constraints. Experimental results demonstrate that NanoMap can judiciously trade off area and delay, and effectively exploit the different features of NATURE.

Categories and Subject Descriptors: B.5.2 [**Design Aids**]: Automatic synthesis
General Terms: Algorithms, design
Keywords: dynamic reconfiguration, logic folding, logic mapping, NATURE, NRAM

1. INTRODUCTION

Since CMOS technology is approaching its physical limits, tremendous efforts are being devoted to nanoscale device and fabrication research [1, 2]. However, since the end of CMOS is still 10-15 years away [3], hybrid nano/CMOS systems will become increasingly attractive in the next few years.

Recent research on reliable nanoscale circuits and architectures has resulted in a variety of nanoelectronic and hybrid nano/CMOS reconfigurable designs. DeHon proposed a nanowire-based programmable logic structure [4]. Snider et al. proposed a defect-tolerant nanoscale fabric using nanowire-based FETs and reconfigurable switches [5]. Strukov et al. proposed CMOL, a hybrid nanowire/CMOS reconfigurable architecture [6]. These designs demonstrate orders of magnitude improvement in performance and integration density. However, they require a self-assembly fabrication process, which is unlikely to be mature in the near future.

Recently, we proposed a hybrid carbon nanotube/CMOS reconfigurable architecture, called NATURE [7]. NATURE 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 run-time reconfiguration. Existing FPGAs allow only a limited number of reconfiguration bits to be stored on-chip because of the area overhead of SRAMs. Since the reconfiguration latency for accessing off-chip storage can be quite large, fine-grain (e.g., cycle-by-cycle) dynamic reconfiguration becomes very difficult [8, 9]. NATURE solves this problem by using non-volatile nanotube RAMs (NRAMs) as on-chip reconfiguration storage. NRAM is a universal memory technology developed by Nantero [10], which is expected to be considerably denser than DRAM and have similar speed to SRAM. Using NRAMs, NATURE improves the capacity of on-chip storage by more than an order of magnitude. Since the access latency of on-chip storage is small, it opens up the opportunity to store multiple logic designs in this storage and invoke different designs through fine-grain dynamic reconfiguration. This leads to an efficient run-time computation model, called temporal logic folding. A large logic circuit can be partitioned into a sequence of logic stages and stored in the on-chip configuration memory. At run-time, stage-by-stage, the logic circuit can be configured into the same hardware and executed in different clock cycles. As reported in [7], using temporal logic folding, logic density can be improved by more than an order of magnitude. Moreover, due to the non-volatile property of NRAM, reconfiguration bits can be maintained in it even when the power supply is switched off. Since these bits do not have to be repeatedly loaded from off-chip memory, this improves the power consumption and security of the system. There exist reconfigurable architectures, such as PipeRench [11], which allow later stages of a pipeline to be executed in the same set of logic blocks that executed an earlier stage. This can be regarded as coarse-grain temporal folding. However, such architectures are largely limited to stream media or DSP applications. NATURE, on the other hand, can support fine-grain temporal folding and does not have these application limitations.

In this paper, we present NanoMap, an integrated design optimization platform for NATURE. NanoMap conducts design optimization from the RTL down to the physical level.

*This work was supported by NSF under Grant No. CCF-0429745.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Given an input design specified in RTL and/or gate-level VHDL, NanoMap optimizes and implements the design on NATURE through logic mapping, temporal clustering, placement, and routing. In the past, numerous design tools have been proposed for reconfigurable architectures with various optimization targets, including area/delay minimization and routability maximization (see [12] for an excellent survey). Our approach differs fundamentally from previous work by targeting the design flexibility enabled by fine-grain temporal logic folding. Given user-specified area/delay constraints, the mapper can automatically explore and identify the best logic folding configuration and make appropriate tradeoffs between circuit delay and area efficiency. It uses a force-directed scheduling (FDS) technique [13] to balance the resource usage across different logic folding cycles. Experimental results demonstrate its efficacy.

2. BACKGROUND

We next provide some background material.

2.1 NATURE architecture

NATURE contains island-style logic blocks, connected by a hierarchical reconfigurable interconnect fabric [7]. Each logic block contains a super-macroblock (SMB) and a local switch matrix connecting the logic block with the interconnect.

2.1.1 SMB architecture

An SMB contains a two-level logic cluster. The first level consists of a set of macroblocks (MBs). Each MB is composed of a set of logic elements (LEs). Multiplexers (low-latency reconfigurable crossbars) are used to form local inter-MB (inter-LE) connections. In NATURE, LE is the atomic functional element. It contains a look-up table (LUT) and flip-flops. Each m-input LUT can realize any m-variable Boolean function. Flip-flops are used to hold computation results used by subsequent cycles.

2.1.2 Support for reconfiguration

NATURE uses NRAMs as on-chip configuration storage. Each individual logic or interconnect element is associated with a k-set NRAM storage. Therefore, k different logic functions can be realized within the same hardware resource. During run-time reconfiguration, reconfiguration bits are read out of NRAMs sequentially, controlled by a counter, and placed into SRAM cells to configure the LEs and switches. A detailed layout and SPICE simulation show that a 16-set NRAM storage introduces 10.6% area overhead with 160ps on-chip reconfiguration time, i.e., the access latency of on-chip NRAM. Under this setup, the logic density is improved by 14X on an average.

2.2 Temporal logic folding

Logic folding enables the realization of different Boolean functions within the same LE in different clock cycles. It can be performed at different levels of granularity, offering significant flexibility in mapping circuits to NATURE. A level-p folding is defined as a setup which requires run-time reconfiguration after the execution of p LUT levels. If p is allowed to be arbitrarily large, it essentially leads to no folding, which is what traditional reconfigurable architectures implement.

Different folding levels result in different circuit delays and area efficiency. Given a logic circuit, increasing the folding level leads to a higher clock period, but smaller cycle count. The overall circuit delay typically decreases as the folding level increases. On the other hand, increasing the folding level also results in much higher resource usage in terms of the number of LEs. Hence, to achieve a desirable area-delay tradeoff, it it crucial to choose an appropriate folding level.

3. MOTIVATIONAL EXAMPLE

In this section, we use an example to demonstrate the design optimization flow of NanoMap. First, we introduce some concepts for ease of exposition. Given an RTL circuit, the registers contained in it are first levelized. The logic between two levels of registers is referred to as a *plane*. 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

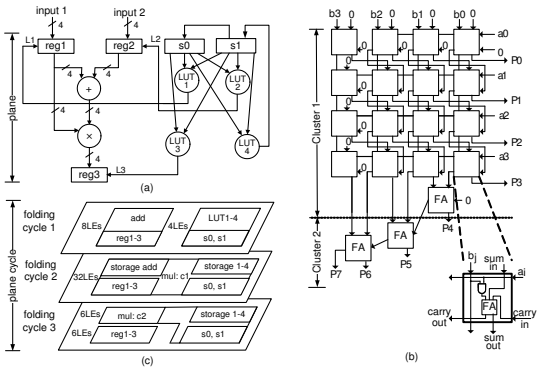

Figure 1: Illustration of (a) an example RTL circuit, (b) module partition, and (c) mapping result

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, i.e., the number of folding stages in one plane, to achieve the best area-delay tradeoff under specified design constraints.

Fig. 1(a) shows an example comprising a four-bit controller-datapath consisting of a single plane. The controller consists of flip-flops s0 and s1 and LUTs LUT1-LUT4, and the datapath consists of registers reg1-reg3, a ripple-carry adder and a parallel multiplier, requiring in all 50 LUTs and 14 flip-flops. The adder consists of eight LUTs with a logic depth, i.e., the number of LUTs along the critical path, of four. The multiplier consists of 38 LUTs with a logic depth of seven. The total logic depth is nine for the plane. Suppose the optimization objective is to minimize circuit delay under a total area constraint of 32 LEs. We assume each LE contains one LUT and two flip-flops. Hence, 32 LEs equal 32 LUTs along with 64 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. As fewer folding stages lead to better circuit delay, NanoMap starts with an initial folding level that results in a minimal number of folding stages and gradually refines it. In the example, the minimal number of folding stages is equal to the total number of LUTs divided by the LUT constraint: $\lceil \frac{50}{32} \rceil = 2$. The initial folding level is then obtained by the maximum logic depth divided by the number of folding stages, which equals $\lceil \frac{9}{2} \rceil = 5$.

Next, based on the chosen folding level, the adder and multiplier are partitioned into a series of connected LUT clusters in 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. Fig. 1(b) shows the partition for the multiplier using level-5 folding. However, the first LUT cluster of the multiplier already needs 34 LUTs, exceeding the area constraint. Thus, the folding level has to be further decreased to level-4 in order to guarantee that each LUT cluster can be accommodated within the available LEs. Correspondingly, the number of folding stages increases to three.

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. If the number of LUTs and flip-flops required by every folding stage is below the area constraint, i.e., 32 LEs, the solution is valid and offers the best possible circuit 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.

Fig. 1(c) shows the mapping result for level-4 folding for the three folding stages. 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 12 LEs, eight LEs for mapping the adder. Four-bit registers reg1-reg3 need two LEs each to accommodate their four flip-flops. They are mapped to the available flip-flops inside the LEs assigned to the adder. Four LEs are also required for LUT1-LUT4 computation. After storing the LUT computation results, there are still four flip-flops left inside the four LEs, some of which can be used to accommodate s0 and s1. Similarly, 32 and 12 LEs are needed for folding cycles 2 and 3, respectively. Folding cycle 2 requires the maximum number of LEs, since multiplier cluster 1, depicted as $mul : c1$, needs 32 LUTs, which occupy 32 LEs. Hence, the number of LEs for mapping this RTL circuit is the maximum required across all the folding cycles, i.e., 32, which is equal to the area constraint.

Next, clustering, which groups LEs into SMBs, placement and routing are performed to produce the final layout of the implementation.

4. NANOMAP: THE OPTIMIZATION FLOW

In this section, we present NanoMap, an integrated design optimization flow developed for NATURE. As shown in Fig. 2, given an input design, NanoMap performs logic mapping, temporal clustering, temporal placement and routing, and produces the configuration bitmap for NATURE.

***Logic mapping*:** (Steps 2–6) 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 [14]. It uses FDS [13] to assign LUTs and LUT clusters to folding stages and balance inter-folding stage resource usage, and produces the LUT network of each temporal folding stage.

***Temporal clustering*:** (Steps 7–8) These steps take the flattened LUT network as input and cluster the LUTs into 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, i.e., LE, MB, or SMB, is temporally shared by logic from different temporal folding stages. Temporal folding necessitates that both intra-stage and inter-stage data dependencies be jointly considered during LUT clustering. Note that the folding stages need not be limited to one plane, i.e., 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 9–14) 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 [15], to provide inter-folding stage resource sharing. Placement is performed in two steps. First, a fast placement is used to derive an initial placement. 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.

In the following sections, we describe the above steps in detail. For logic mapping, we focus on folding level determination and the proposed FDS technique.

4.1 Choosing the folding level

The folding level choice is critical to achieving the best area-delay tradeoff. As we discussed earlier, the best folding level depends on the input circuit structure, which is obtained by identifying each 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: $LUT\_max = max\{num\_LUT_i\},\ i = 1, ..., num\_plane$
- Logic depth of plane i: $depth_i$
- Maximum logic depth among all the planes: $depth\_max = max\{depth_i\},\ i = 1, ..., num\_plane$

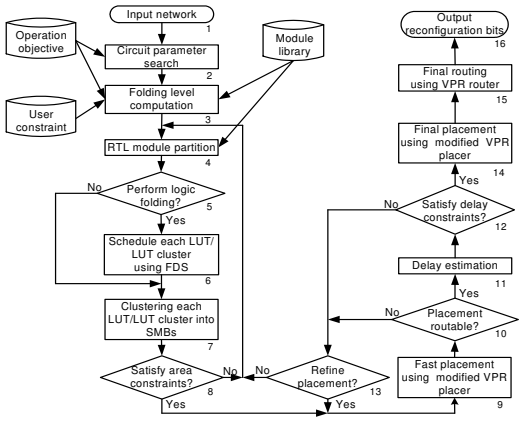

Figure 2: Automatic design flow

- Area constraint, e.g., the available number of LEs: $available\_LE$
- Number of reconfiguration copies in each NRAM: num_reconf

Given the specified optimization objective and constraint, e.g., circuit delay minimization under area constraint or area minimization under delay constraint, etc., the best folding level is computed using above parameters. Due to space limitations, we show how to target one of the design objectives. Similar procedures can target other objectives.

Suppose the optimization goal is to minimize circuit delay. If there is no area constraint, we can use no-folding to obtain the shortest delay, as discussed in Section 2.2. 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*: Since circuit delay is equal to plane cycle times the number of planes in the circuit, plane cycle has to be minimized under the area constraint. First, we stack all the planes together, i.e., resources are shared across all planes, since this does not increase circuit delay but reduces area. Suppose the area used up at this point is LUT_max. If LUT_max is larger than $available\_LE$, logic folding is required to reduce the area within each plane. As discussed in Section 3, the minimum required number of folding stages within each plane is given by:

$$\#folding\_stages = \lceil \frac{LUT\_max}{available\_LE} \rceil \quad (1)$$

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

$$folding\_level = \lceil \frac{depth\_max}{\#folding\_stages} \rceil \quad (2)$$

  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. NanoMap then iterates until the area constraint is met or the folding level reduces to the minimum allowed, min_level, which is limited by num_reconf:

$$min\_level = \lceil \frac{depth\_max * num\_plane}{num\_reconf} \rceil \quad (3)$$

- *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 the folding level requested can be directly computed by the following equation:

$$folding\_level = \lceil \frac{depth\_max * available\_LE}{\sum_i num\_LUT_i} \rceil \quad (4)$$

After an appropriate folding level is chosen, the RTL module is partitioned into LUT clusters accordingly. The original mixed module/LUT network is transformed to an equivalent LUT/(LUT cluster) network which is fed to FDS.

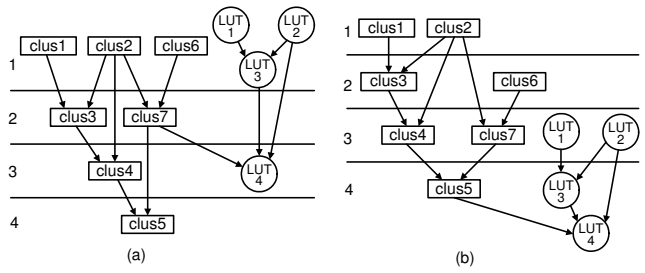

Figure 3: Schedules of LUTs and LUT clusters in a plane: (a) ASAP schedule, and (b) ALAP schedule

4.2 Force-directed scheduling

Different folding stages share the same set of LEs temporally. The overall LE usage is then determined by the folding stage that uses the maximum number of LEs. To optimize overall resource usage, in each plane, we propose to modify FDS [13] to assign the LUT or LUT cluster to folding stages and balance the resource usage of the folding stages.

FDS is a popular scheduling technique in high-level synthesis. However, here we are using it in another scenario. FDS uses an iterative approach to determine the schedule of operations to minimize overall resource usage. Resource usage is modeled as a force. Scheduling of an operation to some time slot, which results in the 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.

In our approach, since the LE usage in each folding cycle is dependent on both the LUT computations and register storage operations performed in parallel, two DGs, one describing the resource usage of the LUT computations and another for register storage usage, have to be built.

4.2.1 Creation of DGs

First, to build the LUT computation DG, the time frame of each LUT or LUT cluster needs to be determined. For a LUT or LUT cluster i, its time frame $time\_frame_i$, or feasible time interval, is 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. From the $ASAP/ALAP$ schedules shown in Fig. 3 for some example, we can see that $time\_frame_{LUT2}$ spans folding cycles 1 to 3, denoted as $[1, 3]$. Here, $clus_i$ denotes LUT cluster i. If a uniform probability distribution is assumed, the probability that this computation is assigned to a feasible folding cycle j within its time frame equals $1/|time\_frame_i|$ for $j \in time\_frame_i$.

A LUT computation DG models the aggregated probability distribution of the potential concurrency of N LUT/(LUT cluster) computations within each folding cycle j, whose value $LUT\_DG(j)$ is the sum of the probabilities of all the computations assigned to this folding cycle, as follows:

$$LUT\_DG(j) = \sum_{i=1}^{N} \frac{1}{|time\_frame_i|} * weight_i \qquad (5)$$

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

To build the register storage DG, which models the distribution of register storage usage, we adopt a similar procedure from [13]. 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. 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. If one or more of the source or destinations are not scheduled, we have to obtain a probabilistic 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 Fig. 4, the output of source computation LUT2, i.e., 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, $|ALAP\_life_{\mathbf{S}}| = 1$.

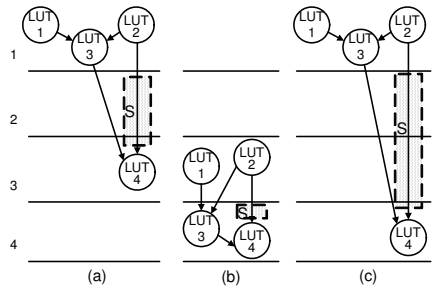

Figure 4: Storage lifetimes for all schedules: (a) ASAP lifetime, (b) ALAP lifetime, and (c) maximum lifetime

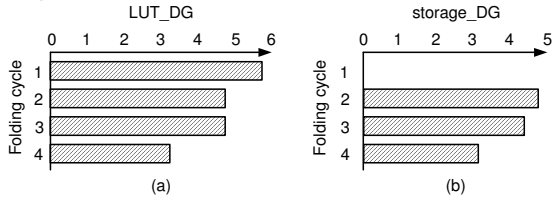

Figure 5: (a) LUT computation DG, and (b) register storage DG for the example in Fig. 3

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:

$$|max\_life| = (ALAP\_life\_end - ASAP\_life\_begin + 1) \qquad (6)$$

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:

$$|overlap| = (ASAP\_life\_end - ALAP\_life\_begin + 1) \qquad (7)$$

Then an estimate of the average length of all possible lifetimes can be obtained by:

$$avg\_life = \frac{|ASAP\_life| + |ALAP\_life| + |max\_life|}{3} \qquad (8)$$

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 $j \in max\_life_i$,

$$storage_i(j) = \frac{avg\_life_i - |overlap_i|}{|max\_life_i| - |overlap_i|} * weight_i \qquad (9)$$

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

$$storage_i(j) = weight_i \qquad (10)$$

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.

$$storage\_DG(j) = \sum_{i=1}^{N} storage_i(j), \quad j \in max\_life_i \qquad (11)$$

The two DGs obtained for the example in Fig. 3 are shown in Fig. 5.

4.2.2 Calculation of forces

In the FDS algorithm, force is used to model the impact of scheduling of operations on resource usage. A higher force implies higher concurrency of run-time operations, which requires more resources in parallel. The force in cycle j is calculated based on DGs, which present the probability of resource usage concurrency:

$$force(j) = DG(j) * x(j) \qquad (12)$$

where $DG(j)$ is either $LUT\_DG(j)$ or $storage\_DG(j)$ in our case, 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:

$$\begin{aligned} self\_force_i(j) &= DG(j) * x(j) + \sum_{k=a,k\neq j}^{b} [DG(k) * x(k)] \\ x(j) &= (|time\_frame_i| - 1)/|time\_frame_i| \\ x(k) &= -1/|time\_frame_i| \end{aligned} \quad (13)$$

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

$$max\{LUT\_self\_force_i(j)/h, storage\_self\_force_i(j)/l\} \quad (14)$$

Assigning a LUT computation to a specific folding cycle will often affect the time frame of its predecessors and successors, which in turn creates additional forces affecting the original move. Equation (13) 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. This results in the least concurrency.

4.2.3 Summary of the FDS algorithm

The pseudo-code of the proposed FDS technique is shown in Algorithm 1. It 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. This procedure continues until all the LUT or LUT cluster computations are scheduled.

Algorithm 1 Force-directed scheduling

```
1: for LUT/(LUT cluster) computations to be scheduled do
2:    evaluate its time frame using ASAP and ALAP scheduling
3:    create the LUT computation and register storage DGs
4:    for each unscheduled LUT/(LUT cluster) computation i do
5:       for each feasible clock cycle j it can be assigned to do
6:          calculate the self-force of assigning node i to cycle j
7:          add predecessor and successor forces to self-forces to get the
            total force for node i in cycle j
8:       end for
9:       select the cycle with the lowest total force for node i
10:   end for
11:   Pick the node with the lowest total force and schedule it in the
      selected cycle
12: end for
```

4.3 Temporal clustering

After scheduling, a network of LUTs is assigned to each folding stage. For each folding stage, we use a constructive algorithm to assign 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. The attraction between a LUT and the seed LUT depends on timing criticality and input pin sharing [16].

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. Such LEs may perform this job over several cycles and feed other LEs in each folding cycle. As illustrated in Fig. 6(a), in an earlier folding cycle, two LUTs may have very few attractions between them (e.g., C and D), but may have a large number of attractions in a later cycle. When performing temporal clustering, the attractions of two LUTs over all the cycles need to be accounted for. Thus, the attraction of such a LUT is set to the maximum of its attractions over all the cycles.

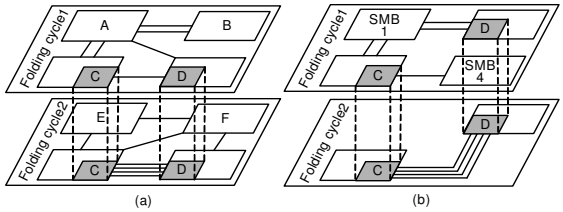

Figure 6: (a) Clustering, and (b) placement with logic folding

4.4 Placement and routing

We modified VPR [15] to perform placement and routing. Placement uses a two-step simulated annealing approach. It starts with a fast low-precision placement. Routability analysis and delay estimation are then used to evaluate the quality of this initial placement. For routability analysis, we use a highly-efficient empirical estimation technique [17]. Delay estimation is based on the timing analysis step of VPR. Routability analysis and delay estimation results are then used to evaluate the feasibility of the initial placement, which determines whether a high-precision placement or another round of logic folding should be invoked.

We modified the VPR placer to support temporal logic folding, which introduces inter-folding stage dependencies. Consider the example in Fig. 6(b). 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. The Manhattan distance is computed between each pair of SMBs belonging to different folding stages. The net bounding box in other unplaced cycles are estimated using this Manhattan distance and added to the cost function for the current cycle to guide placement. Routing by the VPR router is conducted in a hierarchical fashion, first using 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.

4.5 Complexity of the algorithm

In each iteration of the loop, the most computationally intensive steps are FDS and placement, whose complexities are $O(n^2)$ [13] and $O(n^{4/3})$ [18], respectively. The maximum number of iterations performed is related to the maximum logic depth of the circuit. Suppose the logic depth of the circuit is m, then the complexity of the flow is $O(mn^2)$.

5. EXPERIMENTAL RESULTS

In this section, we present experimental results for the mapping of seven RTL/gate-level benchmarks to an instance of NATURE using NanoMap. Based on the observations in [7], we use an architecture instance with one four-input LUT in an LE, four LEs in an MB and four MBs in an SMB to obtain good area-delay trade-offs. 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, as opposed to traditional LEs that include only one flip-flop, we include two flip-flops per LE. This does increase an SMB's area to $1.5X$ (all experiments are based on a 100nm technology). However, this is more than offset by the significant reduction in overall area. 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 tradeoffs when the size of NRAM is fixed to 16 sets instead.

Among the seven benchmarks we targeted, ex1 is the circuit shown in Fig. 1 but with a bit-width of 16. ex2 is an RTL circuit from [19]. Paulin is a differential-equation solver [19], and FIR and Biquad are two types of digital filters. ASPP4 is an application-specific programmable processor [20]. c5315 is

Table 1: Circuit mapping results for AT product optimization

Circuit	#Planes	Max plane depth	#LUTs	#Flip flops	No folding		AT optimization (k enough)				AT optimization ($k = 16$)			
					#LEs	Delay (ns)	Folding level	#LEs	Delay (ns)	AT Improv.	Folding level	#LEs	Delay (ns)	AT Improv.
ex1	1	24	644	50	644	12.90	1	34	17.02	14.36X	2	68	15.60	7.83X
FIR	1	25	678	112	678	14.20	1	56	18.50	9.29X	2	72	16.90	7.91X
ex2	3	22	694	130	694	38.76	1	67	48.84	8.22X	2	88	42.90	7.13X
c5315	1	14	792	0	792	7.86	1	144	10.36	4.17X	1	144	10.36	4.17X
Biquad	1	22	1376	64	1376	12.34	1	68	16.28	15.34X	2	136	14.30	8.73X
Paulin	2	24	1468	147	1468	26.74	1	106	35.52	10.43X	2	136	31.20	9.25X
ASPP4	2	24	2240	160	2240	26.80	1	100	36.96	16.24X	2	200	32.40	9.26X

Table 2: Circuit mapping results for typical optimizations

Circuit	Optimization	Area const. (#LEs)	Delay const. (ns)	Folding level	#LEs	Delay (ns)
ex1	Delay	–	–	1	34	17.02
FIR	Delay	110	–	3	108	16.74
ex2	Area	–	40	11	352	38.04
c5315	Area	–	–	1	144	10.36
Biquad	Delay	100	–	1	68	16.28
Paulin	–	210	30	3	204	29.76
ASPP4	Area	–	28.5	6	600	28.32

a gate-level ALU implementation from the ISCAS'85 benchmark suite. NanoMap was run on a $2GHz$ PC with $1GB$ DRAM under RedHat Linux 9. The mapping CPU times were less than a minute for all the benchmarks.

We first map all benchmarks under the area-time (AT) product minimization objective to show the logic density benefits of temporal logic folding against the traditional no-folding case. The corresponding area (no. of LEs is used as a proxy for area because of the regular architecture), circuit 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 1. We can see that AT product optimization is achieved with folding level-1 in all the cases when there is no restriction on k, because an increase in circuit delay is more than overcome by the dramatic reduction in area. The average reduction in the no. of LEs is $14.8X$ ($9.2X$) and in the AT product $11.0X$ ($7.8X$) at the price of a 31.8% (19.4%) increase in circuit delay for large enough k (k limited to 16).

NanoMap can target many different optimization objectives. Due to space limitations, we choose different optimization objectives for different benchmarks and present the results in Table 2. In Column 2, we mention the objective and in Columns 3 and 4 the constraint (area or delay). 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, we found that global interconnect usage went down by more than 50% when using level-1 folding as opposed to no-folding. This points to trading interconnect area for increased NRAM area as an attractive alternative for NATURE.

6. CONCLUSIONS

In this paper, we introduced an RTL/gate-level automatic design optimization flow, NanoMap, for the hybrid nanotube/CMOS dynamically reconfigurable architecture, NATURE.

NATURE supports fine-grain run-time reconfiguration and, hence, enables logic folding. Through logic folding, significant logic density improvement and flexibility in performing area-delay tradeoffs are made possible. 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 user constraints. With NanoMap, the potential of NATURE can be effectively realized.

7. REFERENCES

[1] Y. Cui, Z. Zhong, D. Wang, W. U. Wang, and C. M. Lieber, "High performance silicon nanowire field effect transistors," *Nano Letters*, vol. 3, pp. 149–152, Jan. 2003.

[2] A. Javey, J. Guo, F. B. Farmer, Q. Wang, and D. Wang, "Carbon nanotube field-effect transistors with integrated ohmic contacts and high-k gate dielectrics," *Nano Letters*, vol. 4, pp. 447–450, Mar. 2004.

[3] "International Technology Roadmap for Semiconductors." http://public.itrs.net

[4] A. Dehon, "3D nanowire-based programmable logic," in *Proc. Int. Conf. on Nano-Networks*, Sept. 2006, pp. 1–5.

[5] G. Snider, P. Kuekes, and R. S. Williams, "CMOS-like logic in defective, nanoscale crossbars," *Nanotechnology*, vol. 15, pp. 881–891, June 2004.

[6] D. B. Strukov and K. K. Likharev, "CMOL FPGA: A reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices," *Nanotechnology*, vol. 16, pp. 888–900, Mar. 2005.

[7] W. Zhang, N. K. Jha, and L. Shang, "NATURE: A hybrid nanotube/CMOS dynamically reconfigurable architecture," in *Proc. Design Automation Conf.*, July 2006, pp. 711–716.

[8] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Adres: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. Field-Programmable Logic and Applications*, Aug. 2003, pp. 61–70.

[9] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. Symp. FPGAs for Custom Computing Machines*, Apr. 1997, pp. 22–28.

[10] "Nantero." http://www.nantero.com

[11] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, pp. 70–77, Apr. 2000.

[12] J. Cong, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Trans. Design Automation of Electronic Systems*, vol. 1, pp. 145–204, Apr. 1996.

[13] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661–679, June 1989.

[14] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table-based FPGA designs," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1–12, Jan. 1994.

[15] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Wkshp. FPGA*, Aug. 1997, pp. 213–222.

[16] A. S. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *Proc. Int. Symp. FPGA*, Feb. 1999, pp. 37–46.

[17] C. L. E. Chang, "RISA: Accurate and efficient placement routability modeling," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 690–695.

[18] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *Proc. Int. Symp. FPGA*, Feb. 2000, pp. 203–213.

[19] L. Lingappan, S. Ravi, and N. K. Jha, "Satisfiability-based test generation for nonseparable RTL controller-datapath circuits," *IEEE Trans. Computer-Aided Design*, vol. 25, pp. 544–557, Mar. 2006.

[20] I. Ghosh, A. Raghunathan, and N. K. Jha, "Hierarchical test generation and design for testability methods for ASPPs and ASIPs," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 357–370, Mar. 1999.