Nonvolatile Memory Allocation and Hierarchy Optimization for High-Level Synthesis

Shuangchen Li *† Ang Li* Yongpan Liu* Yuan Xie† Huazhong Yang*
Dept. of Electronic Engineering, Tsinghua University, Beijing, 100084, China*
Dept. of Electrical and Computer Engineering, University of California at Santa Barbara, CA, 93106, U.S.†
{ypliu,yanghz}@tsinghua.edu.cn*, yuanxie@ece.ucsb.edu†

***Abstract*—The emerging nonvolatile memory (NVM) technology can potentially change the landscape of future IC designs with numerous benefits, such as high performance, low leakage power, and data retention. These advantages motivate designers to exploit utilizing NVM in in ASIC and FPGA. However, unique challenges such as large write energy and asymmetric read/write operations, lead to extra design knobs. This paper focuses on the NVM allocation and hierarchy optimization in high-level synthesis. A hierarchical hybrid memory architecture is presented. The proposed framework optimizes the memory hierarchy, type (NVM or SRAM) and capacity. Both an mixed-integer linear programming (MILP) and a branch-and-bound heuristic are developed. Experimental results demonstrate up to 69.3% power reduction compared with designs without NVM.**

I. Introduction

Recently, the emerging nonvolatile memory (NVM) technology has made a significant impact on the future IC designs. Researchers investigate to use NVMs at different levels in processors. A nonvolatile processor [1] based on NVMs achieved zero-standby power and instant sleep/wakeup for sensor applications. Hybrid nonvolatile cache [2], [3] and scratch pad memory [4] were proposed for high-performance computations.

Utilizing NVM in ASIC and FPGA are also promising. A nonvolatile FIR was fabricated [5] with zero-standby power, and a nonvolatile FFT [6] was proposed for energy harvesting application. In addition, nonvolatile FPGAs were fabricated [7], [8] to achieve small delay and power consumption. Therefore, how to use NVM in future ICs is interesting.

In the past, plenty of research used NVM in processors under given memory hierarchy. Wang *et.al.* proposed a cache management policy dealing with NVM's large write [9]. Data allocation techniques were also proposed in hybrid scratch pad memory [10]. All of them optimized system with fixed memory hierarchy and capacity. However, the NVM synthesis for ASIC and FPGA has not been explored before.

NVM architecture in ASIC and FPGA is quite different from those in microprocessors. It contains many distributed memory blocks and requires the optimization for specific memory access patterns for each of the blocks. The distributed memory architecture provides more flexibility and a larger design space to handle the challenges from NVM devices, such as limited endurance [11], large write energy [12], asymmetric read/write operations [13] and reliability issues [14]. In order to utilize NVMs in application-specific designs, it is necessary to synthesize proper memory architecture for each distributed memory blocks under design constraints. Therefore, a synthesis methodology is needed for NVM-based application-specific hardware designs.

High-level synthesis [15] (HLS) is a good candidate to tackle those challenges. Issenin *et al.* proposed co-synthesis methods for both scratch pad memory and communication architecture for streaming applications [16]. Cong *et al.* [17], [18] combined loop transformation and memory hierarchy allocation in HLS tools to optimize on-chip memory with the bandwidth constraints. However, all previous works did not consider NVM devices, which add new dimensions for memory optimization in HLS.

This paper proposes an approach to optimize and synthesize hybrid NVM and SRAM architectures for ASIC and FPGA designs with HLS tools. The objective is to minimize power consumption under given memory bandwidth, chip area, and performance constraints. The optimization is to decide memory allocations (where to insert a data reuse buffer), memory hierarchy, and memory type (NVM or SRAM) for all distributed memory blocks. Due to high-write energy and asymmetry read/write issues in NVMs, special design knobs are explored. The memory allocation considers both NVM and SRAM options based on memory access patterns. In memory hierarchy, multiple-level hybrid buffer is used to reduce write energy and performance penalty from NVMs. Furthermore, the system models include more parameters to distinguish the memory access patterns due to asymmetric characteristics.

To the best of our knowledge, this is the first work to consider NVM architectures for application-specific designs in high-level synthesis flow, with following contributions.

- We propose a novel hierarchical on-chip memory architecture with hybrid NVM and SRAM (Section II).
- We formulate the optimization as a mixed-integer linear programming (MILP) problem (Section III).
- We develop an efficient branch-and-bound heuristic to solve the problem (Section IV).
- We validate both of the methods showing up to 69.3% power reduction (Section V).

Li and Xie was supported in part by NSF1218867, 1213052 and 1409798. This work was also supported in part by High-Tech Research and Development (863) Program under contract 2013AA01320 and Huawei Shannon Lab and the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions under contract YETP0102.

978-1-4799-7792-5/15/$31.00 ©2015 IEEE

II. Overview

In this section, the proposed hierarchical memory architecture is introduced, followed by a motivation example. Then, the HLS optimization framework is described.

A. Proposed On-chip Memory Architecture

Fig. 1 shows the proposed architecture (not every buffer is necessary). This architecture is adaptable with any synthesizable C program of stream application, and commercial HLS tools [19] are able to synthesize it through proper code motion and script.

Loop kernels in C program are synthesized into *kernels*. It contains processing elements (PEs), which has memory access (*MA*) operations to memories. *Kernels* are connected by FIFOs or ping-pong buffers [20], [21]. *Kernels* work in parallel and form a system pipeline. A hierarchical memory architecture is proposed, consisting of L1 buffers, L2 buffers, and the off-chip main memory. L2 buffers are used between *kernels*, and L1 buffers are used inside the *kernel*. All the buffers can be implemented by either NVM or SRAM. In addition, they are all optional: If L2 buffers do not exist, L1 buffers or PEs access the off-chip memory directly; If L1 buffers do not exist, PEs direct access L2 or the off-chip memory. Specifically, *self-reuse* and *group-reuse* are two types of L1 buffer: *self-reuse* buffers data of MA_i for its own usage; *group-reuse* buffers data that MA_i and MA_j share. The two types of buffers are also both optional, and form a micro-hierarchy: when they co-exist, *self-reuse* is used as a lower-level buffer for *group-reuse*.

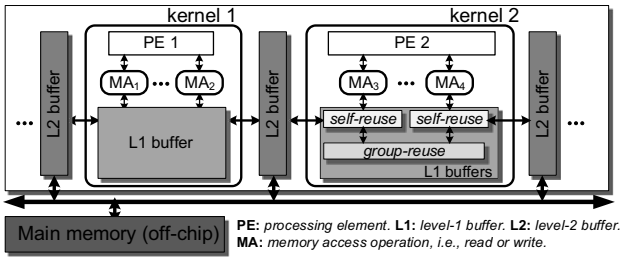

Fig. 1. Proposed hierarchical memory architecture

The proposed architecture has special considerations for NVM: It considers data reuse of both *inner*-loop kernel (L1 buffers) and *inter*-loop kernel (L2 buffers). In addition, it considers both *self-reuse* and *group-reuse* cases in data reuse, different from existing works [18] considering *group-reuse* only. The *self-reuse* usually appears in write-intensive situations, where NVMs need special considerations.

B. A Motivation Example

Table I shows the parameters of 512KB NVM and SRAM from NVSim [22]. Compared with the same-size SRAM, NVMs have ultra-low leakage power and small area but large write energy and latency. The following example shows the motivations to do the memory architecture optimization.

Fig. 2(a) shows a C program segment from *Term Analysis* application (in Section V). Fig. 2(b) to (d) show three designs

TABLE I
The parameters of a 512KB NVM and SRAM

Type	Latency (ns)		Energy (pJ)		Leakage (μW)	Area (mm^2)
	read	write	read	write		
SRAM	13.7	13.7	18.0	11.6	148.7	0.78
MRAM	14.3	17.4	12.3	34.8	4.7	0.26

with different memory architectures. Table II compares the power consumption in the three cases. Fig. 2(b) shows a design with a single L2 SRAM buffer, which suffers from high leakage power. In order to reduce it, we replace the L2 SRAM buffer by an NVM buffer in Fig. 2(c). However, the replacement turn out to increase the power consumption by 128.5% (see Table II), because *kernel 1* is write intensive. Large write energy of NVMs overwhelms its advantage of low leakage. Fig. 2(d) shows the optimal design, where a hierarchical memory architecture with hybrid buffer types is adopted. L1 SRAM buffer reduces write operations to L2 NVM buffer from 2^{30} to 2^{19}. Therefore, it saves 88.45% power. This motivation example shows that simply replacing all SRAM buffers with NVM ones may not be a good choice. It is necessary to carefully explore memory architectures, such as allocation, hierarchy, type and size in a synthesis framework.

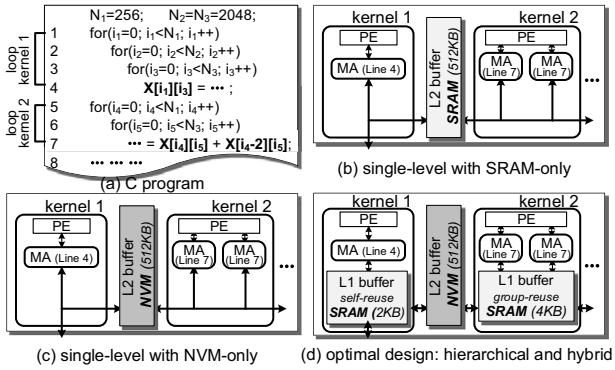

Fig. 2. An motivation example with different memory architectures

TABLE II
Power consumption under different architectures

	Design choice	Write count to NVMs	On-chip buffer power	Power saved
Fig. 2(b)	single-level, SRAM-only	–	0.61mW	–
Fig. 2(c)	single-level, NVM-only	2^{30}	1.40mW	**-128.5%**
Fig. 2(d)	hierarchical, hybrid	2^{19}	0.07mW	**88.45%**

C. HLS Framework to support NVMs

Fig. 3 shows the HLS framework. The framework's inputs are a synthesizable C program, design constraints, and platform parameters. The target platform can be either ASIC or FPGA with both NVM and SRAM available. Both an MILP and a heuristic solution are proposed, which solve the hierarchy and allocation of the buffers, the buffer types, and size. The selection of either of the two method depends user's requirement for optimality and running time. The results are

provided to a code generator, which generates the modified C program segments with scripts. Finally, all these codes are synthesized by a commercial HLS tool.

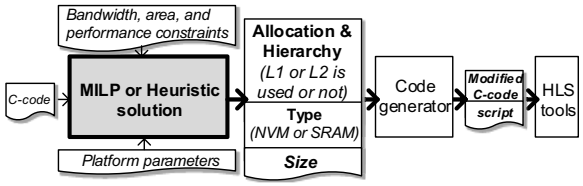

Fig. 3. Proposed high-level synthesis framework to support NVMs

III. MILP Formulation

In this section, the NVM allocation and hierarchy problem is formulated as a mixed-linear programming (MILP) problem.

A. Parameters and Variables Definitions

Table III (upper part) shows the parameters. An s-size (e.g., 8KB) RAM's dynamic energy $\vec{E}_s$, latency $\vec{L}_s$, leakage power $\vec{P}_s$, and area $\vec{A}_s$ are all 1×2 vectors containing both NVM's and SRAM's parameters, e.g., $\vec{E}_s^W$=[$E_s^{\text{W,nvm}}$, $E_s^{\text{W,sram}}$]. They come from NVSim [22]. $S_{i,j}$, S_k represent the buffer size, and C_i, C_i^S represent the access count. They come from C program analysis. If the C program can be presented by the polyhedral model [23], [24], these parameters are calculated by methods from compiler theory. Otherwise, they are extracted by run-time statistic. BW_{const}, A_{const}, CLK_{const} are user defined bandwidth, area, and performance constraints. Table III (lower part) shows the variables. The vector $\vec{x}_{i,j}$=[$x_{i,j}^{\text{nvm}}$, $x_{i,j}^{\text{sram}}$]$^\intercal$ represents the design option of L1 buffer between MA_i and MA_j, e.g., if $x_{i,j}^{\text{nvm}}$=1, L1 buffer is NVM. The vector $\vec{y}_k$=[y_k^{nvm}, y_k^{sram}]$^\intercal$ represents the design option of L2 buffer behind $kernel_k$.

TABLE III
Parameter and Variable Definition

	Notation	Description
Parameters	$\{\vec{E}_s^W\}$, $\{\vec{E}_s^R\}$, $\{\vec{L}_s^W\}$, $\{\vec{L}_s^R\}$	$\mathbb{R}^{1\times2}$ vectors; Write (read) operation energy (μJ) or latency (μs) of s-size [NVM, SRAM]
	$\{\vec{P}_s\} \in \mathbb{R}^{1\times2}$	Leakage of s-size RAM, [P_s^{nvm}, P_s^{sram}] (μW)
	$\{\vec{A}_s\} \in \mathbb{R}^{1\times2}$	Area of s-size RAM, [A_s^{nvm}, A_s^{sram}] (μm^2)
	E_{off}^W, E_{off}^R	Write (read) energy of the off-chip memory (μJ)
	$\{S_{i,j}\}$,$\{S_k\}$	L1 size from MA_i to MA_j, L2 size behind $kernel_k$
	$\{C_i\}$, $\{C_i^S\}$	Access count of MA_i, see Section III-B1 for detail
	T	Total execution time of the system (s)
	BW_{const}	Off-chip bandwidth(Byte/s)
	A_{const}	On-chip memory area constraint (μm^2)
	CLK_{const}	Performance constraint, target clock (MHz)
Variables	$\{\vec{x}_{i,j}\} \in \mathbb{B}^{2\times1}$	L1 from MA_i to MA_j is NVM ($\vec{x}_{i,j}$=[1, 0]$^\intercal$), or SRAM ($\vec{x}_{i,j}$=[0, 1]$^\intercal$), or not exists ($\vec{x}_{i,j}$=[0, 0]$^\intercal$)
	$\{\vec{y}_k\} \in \mathbb{B}^{2\times1}$	L2 behind $kernel_k$ is NVM ($\vec{y}_k$=[1, 0]$^\intercal$), or SRAM ($\vec{y}_k$=[0, 1]$^\intercal$), or not exists ($\vec{y}_k$=[0, 0]$^\intercal$)

B. MILP Formulation

Given the parameters in Table III, we need to solve the optimal NVM allocation and hierarchy ($\{\vec{x}_{i,j}\}$ and $\{\vec{y}_k\}$). The objective is to minimizing memory power consumption, while meeting the bandwidth, area, and performance constraints. The specific formulation is described as Equation (1). The power consumption contains dynamic power of each MA_i's L1 buffers, each $kernel_k$'s L2 buffer, and the off-chip memory, as well as their leakage power.

$$\text{obj.: } \min . \sum_{\forall i} p_{\text{dyn}}^{\text{L1}}(i) + \sum_{\forall k} p_{\text{dyn}}^{\text{L2}}(k) + p_{\text{dyn}}^{\text{off}} + p_{\text{leak}} \tag{1}$$

$$\text{s.t.: } area < A_{\text{const}} \quad bw < BW_{\text{const}} \quad delay_{\text{crit}} < CLK_{\text{const}}^{-1}$$

Each expression in the formulation is described below.

1) L1 buffer's dynamic power $p_{dyn}^{L1}(i)$: Self-reuse (using $\vec{x}_{i,i}$ to solve) and *group-reuse* ($\vec{x}_{i,j}$) are two types of L1 buffer, which are calculated separately. Equation (2a) shows *self-reuse* buffer's power $sf(i)$ (MA_i write case). The $S_{i,i}$-size buffer's parameter $\vec{E}_{S_{i,i}}^R$ and $\vec{E}_{S_{i,i}}^W$ are multiplied by $\vec{x}_{i,i}$ to select the type (NVM or SRAM or none), which is then multiplied by access count C_i or C_i^S. C_i is MA_i's access count to its first connected memory; C_i^S is MA_i's *self-reuse* buffer's access count to higher-level memories. Since temporary data are stored in *self-reuse* buffer, and only the final results are written back to higher-level buffers, $C_i \geq C_i^S$ always holds.

$$p_{\text{dyn}}^{\text{L1}}(i) = sf(i) + gp(i), \tag{2}$$

$$sf(i) = (\vec{E}_{S_{i,i}}^W \vec{x}_{i,i} \cdot C_i + \vec{E}_{S_{i,i}}^R \vec{x}_{i,i} \cdot C_i^S)/T, \tag{2a}$$

$$gp(i) = (\sum_{\forall m} \vec{E}_{S_{i,m}}^W \vec{x}_{i,m} \cdot \widetilde{c}_i + \sum_{\forall m} \vec{E}_{S_{m,i}}^R \vec{x}_{m,i} \cdot \widetilde{c}_i)/T, \tag{2b}$$

$$\text{where} \quad \widetilde{c}_i = (1 - ||\vec{x}_{i,i}||) \cdot C_i + ||\vec{x}_{i,i}|| \cdot C_i^S. \tag{3}$$

Equation (2a) shows *group-reuse*'s power $gp(i)$. If there is no *self-reuse* ($||\vec{x}_{i,i}||$=0), the access count is C_i. Otherwise, it is C_i^S. $\widetilde{c}_i$ in Equation (3) calculates this.

2) L2 buffer's dynamic power $p_{dyn}^{L2}(k)$: Binary variables $\{z_{k,i}^W\}$ are introduced to capture writing operation from MA_i to $kernel_k$ ($z_{i,k}^R$ for reading, respectively) through either L2 buffer or off-chip memory. $\{z_{k,i}^W\}$ are calculated by $\vec{x}_{i,j}$ and $\vec{y}_k$, e.g., if L1 buffer exists, there is no need for MA_i to access other $kernel_k$ ($z_{k,i}^W$=0). Therefore, $p_{\text{dyn}}^{\text{L2}}(k)$ is the summary of the operation ($z_{i,k}^W$, $z_{k,i}^R$) multiplied by byte access energy and access count $\widetilde{c}_m$, as follows,

$$p_{\text{dyn}}^{\text{L2}}(k) = (\sum_{\forall i} z_{k,i}^W \cdot \vec{E}_{S_k}^W \vec{y}_k \cdot \widetilde{c}_i + \sum_{\forall m} z_{i,k}^R \cdot \vec{E}_{S_k}^R \vec{y}_k \cdot \widetilde{c}_i)/T, \tag{4}$$

3) Off-chip memory's dynamic power p_{dyn}^{off}: If L2 buffer exists ($||\vec{y}_k||$=1), there is no access to off-chip memory and hence no power consumed. Otherwise, $p_{\text{dyn}}^{\text{off}}$ is the summary of each off-chip operation's ($z_{i,k}^W$, $z_{k,i}^R$) power consumption (byte energy multiplying access count $\widetilde{c}_m$ divided by time T).

$$p_{\text{dyn}}^{\text{off}} = \sum_{\forall i,k} (1 - ||\vec{y}_k||)(z_{i,k}^W E_{\text{off}}^W \widetilde{c}_i + z_{k,i}^R E_{\text{off}}^R \widetilde{c}_i)/T. \tag{5}$$

4) On-chip leakage power p_{leak} and area $area$: The S-size leakage power or area parameter are selected by the variables to choose the type (NVM or SRAM or none), as follows,

$$p_{\text{leak}} = \sum_{\forall i,j} \vec{P}_{S_{i,j}} \cdot \vec{x}_{i,j} + \sum_{\forall k} \vec{P}_{S_k} \cdot \vec{y}_k, \tag{6}$$

$$area = \sum_{\forall i,j} \vec{A}_{S_{i,j}} \cdot \vec{x}_{i,j} + \sum_{\forall k} \vec{A}_{S_k} \cdot \vec{y}_k. \tag{7}$$

5) Bandwidth bw: Bandwidth is defined as average off-chip access count in T. If L2 buffer exists ($||\vec{y}_k||$=1), there is no access to off-chip memory. Otherwise, the bandwidth is $z_{i,k}^W$ and $z_{k,i}^R$ (representing connection between *MA* and off-chip memory when $||\vec{y}_k||$=0) multiplying the access count $\widetilde{c}_i$.

$$bw = \sum_{\forall k} (1 - ||\vec{y}_k||) \cdot (\sum_{\forall i} z_{i,k}^W \cdot \widetilde{c}_i + \sum_{\forall i} z_{k,i}^R \cdot \widetilde{c}_i)/T, \tag{8}$$

6) Critical delay $delay_{crit}$: Since the logic delay is fixed, only memory access latency is considered. The critical delay is the largest accessing delay of the memory, as follows,

$$delay_{\text{crit}} = \max\{\vec{L}^R_{S_{i,j}}\vec{x}_{i,j},\ \vec{L}^W_{S_{i,j}}\vec{x}_{i,j},\ \vec{L}^R_{S_k}\vec{y}_k,\ \vec{L}^W_{S_k}\vec{y}_k\}. \quad (9)$$

To sum up, the formulation includes Equation (1) to (9). There are also several consistent constraints between variables, which is handled by a data reuse graph [24]. The multiplication between binary variables in Equation (2) to (8) can be transformed into linear forms by mathematical transformations [25]. Therefore, this is an MILP problem.

IV. Heuristic Solution

A heuristic solution is a good candidate when the MILP is too costly. In this section, the NVM allocation problem is generalized as a variation of the knapsack problem. A branch-and-bound heuristic is then proposed to solve it.

A. The knapsack problem and challenges

In order to generalize the NVM optimization as a knapsack problem, a higher-level formulation is abstracted from MILP formulation in Equation (1), as follows,

$$\max \sum_i \Delta p_i^{\text{save}} \cdot x_i,\ \text{s.t.}\ \sum_i \Delta bw_i^{\text{save}} x_i > BW_{\text{const}}^{\text{save}}\ \sum_i \Delta a_i x_i < A_{\text{const}},$$

where x_i is 0-1 variable, denoting the selection of a buffer. The problem is regarded as a variation of 0-1 knapsack problem: when adding a buffer to the system, it *saves* Δp_i^{save} power, *saves* $\Delta bw_i^{\text{save}}$ bandwidth, and *costs* Δa_i. The goal is to maximize power saving (total items' value) while satisfying bandwidth and area constraint. Note that buffers violating performance constraints are not picked,, which satisfies the performance constraint.

Although regarded as a knapsack problem, it is still difficult (NP-hard) to solve, especially in this NVM allocation problem. Different from standard knapsack problems, there are three unique challenges.

1) Time-variant item value: The buffer's power saving Δp_i^{save} is time-variant. It depends on the current memory architecture. Anytime the heuristic selects or discards one buffer, the architecture changes and all the Δp_i^{save} needs updating. It makes the classic 0-1 knapsack problem solution (greedy or dynamic programming method) not adoptable. Therefore, a branch-and-bound algorithm is proposed. Experiment result in Table V shows the impact of this challenge.

2) Mutually-exclusive items: The NVM and SRAM buffer at the same location are exclusive. This leads to another design knob: when a solution still have area slack but adding any other buffer fails to improve power saving, exchanging an NVM with SRAM in the current solution might save more power.

3) Multiple constraints: Bandwidth saved and area consumed are two independent constraints. Moreover, they take place in upper bound and lower bound, respectively. This challenge leads to decision complexity in the branch-and-bound algorithm.

B. The branch-and-bound based heuristic

Algorithm 1 shows the heuristic. It begins with a primary result (Line 1 to 8), and then the branch-and-bound function `expBB` improves the result (Line 10). The primary result is found by selecting buffers by the large of $\Delta p_i^{\text{save}}/\Delta a_i$, until no buffers improve power saving or area constraint is violated (Line 6). The following branch-and-bound function `expBB` is shown in Algorithm 2.

Algorithm 1: The branch-and-bound heuristic (main body)

```
   input : parameters of M buffer candidates, constraints
   output: the result {x}
 1 initial {x}=∅, update p_now^save, bw_now^save, a_now by {x};
 2 initial the sequence Sq_p by sorting (Δp_i^save/Δa_i);
 3 initial the sequence Sq_bw by sorting (Δbw_i^save/Δa_i);
 4 while k < M do
 5     update Sq_p and Sq_bw considering current {x};
 6     get buffer i corresponding to the k-th large value in Sq_p;
 7     if Δp_i^save < 0 or a_now > A_const then break;
 8     else add x_i to {x}, k++;
 9 end
10 if expBB({x}, k, k) = true then return {x} ;
11 else return fail ;
```

The brand-and-bound function `expBB` (Algorithm 2) is a recursive procedure. Its inputs represent that if buffer i corresponds to the k^{th} largest value in Sq_{p} and $s<k\leq t$ holds, x_i is fixed to 0 or 1. At each recursion, the heuristic adds (or remove) one buffer i (k^{th} in Sq_{p}) with $k>t$ (or $k\leq s$) to explore better solutions, and the range of $[s,t]$ gets extended. A bound is added to avoid enumerating every branch ($(t,M]$ or $[1,s]$) at each recursion. Specifically, if there is area margin, the heuristic adds one buffer to the current result (Line 3 to 16) Otherwise, it removes one buffer (Line 17 to 23). When area is slack, it explores to add *one* buffer from $[t,M]$ in Sq_{p} (Line 6 to 8) until the bound is reached (Line 5). The `bound` is defined as $\Delta a_i/(A_{\text{const}}-a_{\text{now}})<\alpha$, where α is a user defined factor. If no more buffers from $[t,M]$ in Sq_{p} provides power saving (Line 9 to 11), it explore Sq_{bw} instead, in order to meet the bandwidth constraint. If the bandwidth constraint is also met (Line 12 to 14), it exchanges the NVM in the current result with SRAM to further explore better solutions.

C. Algorithm Analysis

The branch-and-bound function has special design for the three challenges mentioned in Section IV-A. First, in order to tackle the time-variant item value challenge, the heuristic updates Sq_{p} and Sq_{bw} at each recursion (Line 1). Moreover, the `bound` (Line 4 and 18) needs special calculation. Since adding any buffer make other buffer's Δp_i^{save} smaller, bound defined in an exact knapsack problem algorithm [26] is slack in this particular problem. In older to shorten the run time, this heuristic uses area as bound. Second, it tackles the mutually-exclusive item challenge by adding NVM/SRAM design space exploration (Line 12). Result in Table VI shows that it reduces the error by 242%. Third, to deal with multiple constraints, it adds extra decision block (Line 8 to 11).

The heuristic's complexity is $O(2^B logM)$, when there are M possible buffers and the branch is bounded as B. B is

Algorithm 2: The branch-and-bound expBB($\{x\}$,s,t)

input : $\{x\}$, buffers from s to t in Sq_{p} are fixed
output: the result $\{x\}$
initial $improve$ = false, $p_{\text{now}}^{\text{save}}$, $bw_{\text{now}}^{\text{save}}$, a_{now} by $\{x\}$;
if $a_{now}<A_{const}$ **then**
  for $t_+ = t$ *to* M **do**
    if *reach the* bound **then return** $improve$;
    if $\Delta p_i^{save} > 0$ **then**
      get buffer i with the t_+^{th} large value in Sq_{p};
      $improve$ |= expBB($\{x\} \cup x_i$, s, t_+);
    else if $bw_{now}^{save}<BW_{const}^{save}$ **then**
      get buffer i with the t_+^{th} value in Sq_{bw};
      $improve$ |= expBB($\{x\} \cup x_i$, s, t_+);
    else
      explore by exchanging NVM with SRAM;
      if *get better solution* **then** update $\{x\}$ and **return** true;
    end
  end
else
  for $s_- = s$ *to* 1 **do**
    if *reach the* bound **then return** $improve$;
    get buffer i with the s_-^{th} large value in Sq_{p};
    $improve$ |= expBB($\{x\} \setminus x_i$, s_-, t);
  end
end

very small and the experimental results in Section V-B show the efficiency.

V. Experiment

This section starts with the introduction of the experiment setup. It then shows both the superior and the flexibility of our method. At last, the analysis of the heuristic is shown.

A. Experiment Setup

The benchmarks are streaming applications for image processing and data mining. They are all data-intensive and contain multiple loop kernels. The detail are listed as follows,

- *Hist Equal*: an image histogram equalization algorithm.
- *Watermarking*: watermarking with DCT and IDCT.
- *Marr Edge*: an image Marr-Hildreth edge-detecting.
- *Term Analysis*: analyzing the term frequency.

Parameters of both SRAM and NVM (STT-RAM) come from NVSim [22] (32 nm technology), since the model in NVSim is adaptable for both embedded RAM in ASIC and BRAM in FPGA. The off-chip main memory is a 128M phase-change memory [27]. Parameter T (hardware total execute time) are evaluated by applying high-level synthesis tool Vivado HLS [19], since the selection of SRAM or NVM have no impact on execution cycles. In the optimization, the MILP is solved by Lingo [28]. The code generator are temporally implemented manually, which can be automatically generated by applying tools like ClooG [29].

B. Superior of the Proposed Method

In order to show the superior of the proposed method, the results are compared with other designs. Fig. 4 shows the comparison using the four benchmarks. The first four bars in each benchmark cluster are the suboptimal designs. They miss one or both design knobs of the memory hierarchy and type selection. Specifically, the *L2-only SRAM-only* (*L2-only NVM-only*) is a design with only L2 SRAM (NVM) buffers. The *hierarchical SRAM-only* (*hierarchical NVM-only*) is a design with both potential L1 and L2 SRAM (NVM) buffers. The last bar is the result of the proposed method. All the power consumptions are normalized.

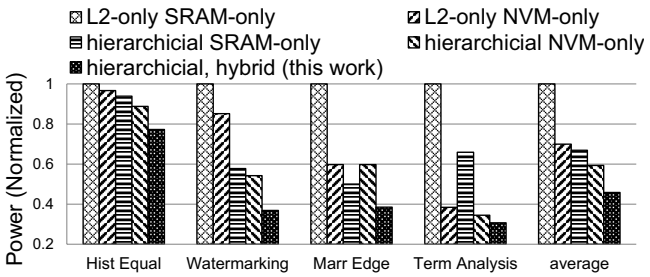

Fig. 4. The comparison with naive designs for the four benchmarks

Fig. 4 show that, utilizing NVM (including *L2-only NVM-only* and *hierarchical NVM-only*) saves 37.6% power on average than traditional design with only SRAM (including *L2-only SRAM-only* and *hierarchical SRAM-only*). This demonstrates that utilizing NVM in application-specific hardware design is promising since it provides the small area and leakage power consumption. However, the buffer optimization is nontrivial. Simple NVM-only design might turn out to cost more power, such as the *hierarchical NVM-only* and *hierarchical SRAM-only* in *Marr Edge* benchmark with 19.3% extra power consumption. It is because the NVM has large write energy, which is sensitive with write-intensive application. This is why the proposed optimization is necessary.

Fig. 4 also show that the proposed method are always better. It saves up to 69.3% power consumption in *Term Analysis*. On average, the proposed method saves 54.2%, 24.2%, 21.1%, and 13.5% power than these four suboptimal designs, respectively. These results indicate that, single memory allocation optimization is not enough. Both memory hierarchy and hybrid memory type are required to utilize NVMs in hardware designs.

C. Flexibility of the Proposed Method

Table IV illustrates optimization results under different design constraints of the *Hist Equal* benchmark. It validates the proposed method is flexible to explore a large design space. Row 1 and 2 show the impact of the area constraint. When the area constraint is shrunk from 40000 to 35000, the system has to use more NVMs in order to save area, even in some write intensive situations. This leads to extra power consumption. Row 3 and 4 show the impact of the bandwidth constraint. When it is shrunk from 2000 to 1500, more on-chip buffers are added to save off-chip bandwidth and hence cost extra power and area consumption. Row 5 and 6 show the impact of performance constraint. When the target clock frequency raises from 100MHz to 150MHz, some NVMs are discarded because of large write latency. Therefore, the system has to use SRAMs instead, which leads to extra power and

area consumption. Table IV also shows that, whatever the constraints are, the proposed method is capable of finding the optimal result, which perfectly meets these constraints.

TABLE IV
OPTIMIZING *Hist Equal* UNDER DIFFERENT CONSTRAINTS

Constraints			Results		
CLK_{const}	BW_{const}	A_{const}	bw	area	power
100MHz	4000	40000	3840	37231	15.90
		35000	2304	26137	20.42
100MHz	2000	60000	1536	57231	19.45
	1500		768	59027	19.61
100MHz	3000	100000	2304	48129	17.75
125MHz			2304	97203	22.67

D. Analysis of the Heuristic Solution

Heuristic solution is a good candidate when the MILP is too costly. Table V shows the comparison of the two solution. Besides the four benchmarks, two large-scale random cases are also included. The MILP finds the optimal result in seconds encountering small variable's numbers (Column 2 to 5). As the variable's number increases, however, the MILP's run time grows extremely large, which even leads to failure (Column 7). In these cases, the heuristic solution still finds the optimal result in milliseconds.

TABLE V
COMPARISON BETWEEN THE MILP AND THE HEURISTIC SOLUTION

	Term	Marr	Hist	Water	Rand 1	Rand 2
variables#	100	120	127	198	686	2641
MILP (ms)	2×10^3	2×10^3	1×10^3	3×10^3	1528×10^3	*failure*
Heu. (ms)	5.39	6.20	6.43	8.27	2.70	2.99

Table VI shows the heuristic's error analysis. Row 2 shows the error if branch-and-bound function `expBB` is absent. It could cost 123.5% error or even failure, which shows the importance of the branch-and-bound algorithm. Row 3 shows the error if Line 13 to 14 is removed from Algorithm 2. It could lead to 242.8% error. This shows that, dealing with the special challenge (mutually-exclusive item) of the problem is essential. Row 4 shows the error if time-variant item value is not considered. It could cost 148% error and even failure. This is because the branch-and-bound function fails to improve the result. Incorrect Δp_i^{save} leads to incorrect decisions when exploring branches. It risks missing branches inside the bounds. This shows that, dealing with the time-variant item value challenge is essential.

TABLE VI
ERROR ANALYSIS OF THE BRANCH-AND-BOUND HEURISTIC

	Term Analysis	Hist Equal	Marr Edge	Watermarking
error (w.o. BB)	123.5%	*failure*	30.6%	242.8%
error (w.o. EX)	0%	13.7%	30.6%	242.8%
error (constant Δp_i^{save})	148.9%	*failure*	5.41%	29.2%
final error	0%	0%	0%	0%

VI. CONCLUSION

Emerging NVM has the great potentials to replace SRAM or DRAM memory due to many benefits. This paper proposes a HLS framework to optimize the memory architecture for FPGA/ASIC. Both an MILP and a heuristic solution are proposed. The experimental results show 69.3% power reduction.

REFERENCES

[1] Y. Wang, Y. Liu, and *et al.*, "A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops," in *ESSCIRC*, 2012, pp. 149–152.
[2] G. Sun, Y. Joo, and *et al.*, "A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement," in *HPCA*, 2010, pp. 1–12.
[3] Y. Chen and *et al.*, "Processor caches built using multi-level spin-transfer torque ram cells," in *ISLPED*. IEEE, 2011, pp. 73–78.
[4] P. Wang, "Designing scratchpad memory architecture with emerging stt-ram memory technologies," in *ISCAS*, 2013, pp. 1244–1247.
[5] J. Zhao and *et al.*, "Kiln: closing the performance gap between systems with and without persistence support," in *MICRO*, 2013, pp. 421–432.
[6] A. Mirhoseini and *et al.*, "Automated checkpointing for enabling intensive applications on energy harvesting devices," in *ISLPED*, 2013.
[7] T. Aoki, Y. Okamoto, and *et al.*, "Normally-off computing with crystalline ingazno-based fpga," in *ISSCC*, 2014, pp. 502–503.
[8] Y. Y. Liauw and *et al.*, "Nonvolatile 3d-fpga with monolithically stacked rram-based configuration memory," in *ISSCC*, 2012, pp. 406–408.
[9] J. Wang and Y. Xie, "Oap: an obstruction-aware cache management policy for stt-ram last-level caches," in *DATE*, 2013, pp. 847–852.
[10] J. Hu, C. J. Xue, and *et al.*, "Data allocation optimization for hybrid scratch pad memory with sram and nonvolatile memory," *TVLSI*, vol. 21, no. 6, pp. 1094–1102, 2013.
[11] J. Wang, "i2wap: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations," in *HPCA*, 2013, pp. 234–245.
[12] X. Dong, X. Wu, and *et al.*, "Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement," in *DAC*, 2008, pp. 554–559.
[13] G. Sun, X. Dong, and *et al.*, "A novel architecture of the 3d stacked mram l2 cache for cmps," in *HPCA*, 2009, pp. 239–249.
[14] J. Wang and *et al.*, "Point and discard: a hard-error-tolerant architecture for non-volatile last level caches," in *DAC*, 2012, pp. 253–258.
[15] J. Cong, B. Liu, and *et al.*, "High-level synthesis for fpgas: From prototyping to deployment," *TCAD*, vol. 30, no. 4, pp. 473–491, 2011.
[16] I. Issenin and N. Dutt, "Data reuse driven energy-aware mpsoc co-synthesis of memory and communication architecture for streaming applications," in *CODES+ISSS*, 2006, pp. 294–299.
[17] L.-N. Pouchet, P. Zhang, and *et al.*, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA*, 2013, pp. 29–38.
[18] J. Cong and *et al.*, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *DAC*, 2012, pp. 1233–1238.
[19] "Vivado high-level synthesis - xilinx electronic system level design." [Online]. Available: http://www.xilinx.com/
[20] S. Li, Y. Liu, and *et al.*, "Optimal partition with block-level parallelization in c-to-rtl synthesis for streaming applications." in *ASP-DAC*, 2013, pp. 225–230.
[21] H. Javaid, X. He, and *et al.*, "Optimal synthesis of latency and throughput constrained pipelined mpsocs targeting streaming applications," in *CODES+ISSS*, 2010, pp. 75–84.
[22] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, vol. 31, no. 7, pp. 994–1007, 2012.
[23] W. Li, "Compiling for numa parallel machines," Cornell University, Tech. Rep., 1994.
[24] "Nonvolatile memory allocation and hierarchy optimization for high-level synthesis," 2014. [Online]. Available: http://nics.ee.tsinghua.edu.cn/people/lisc10/NVHLS_TR.pdf
[25] F. Glover, "Improved linear integer programming formulations of nonlinear integer problems," *Management Science*, vol. 22, no. 4, 1975.
[26] D. Pisinger, "An expanding-core algorithm for the exact 0–1 knapsack problem," *European Journal of Operational Research*, vol. 87, no. 1, pp. 175–187, 1995.
[27] "Micro – p8p parallel phase change memory." [Online]. Available: http://www.micron.com/
[28] "Lingo 14.0 - optimization modeling software for linear, nonlinear, and integer programming." [Online]. Available: http://www.lindo.com/
[29] "The cloog code generator." [Online]. Available: http://www.cloog.org/