Joint Precision Optimization and High Level Synthesis for Approximate Computing

Chaofan Li[*], Wei Luo[*], Sachin S. Sapatnekar[+] and Jiang Hu[*]
[*]Department of ECE, Texas A&M University
[+]Department of ECE, University of Minnesota
{chaof,lw-1022}@tamu.edu; sachin@umn.edu; jianghu@ece.tamu.edu

ABSTRACT

Approximate computing has been recognized as an effective low power technique for applications with intrinsic error tolerance, such as image processing and machine learning. Existing efforts on this front are mostly focused on approximate circuit design, approximate logic synthesis or processor architecture approximation techniques. This work aims at how to make good use of approximate circuits at system and block level. In particular, approximation aware scheduling, functional unit allocation and binding algorithms are developed for data intensive applications. Simple yet credible error models, which are essential for precision control in the optimizations, are investigated. The algorithms are further extended to include bitwidth optimization in fixed point computations. Experimental results, including those from Verilog simulations, indicate that the proposed techniques facilitate desired energy savings under latency and accuracy constraints.

Keywords

High level synthesis, approximate computing

1. INTRODUCTION

Near the end of Moore's law, the advances of VLSI technology are increasingly constrained by the fundamental limit of chip power density. As such, almost any opportunity for improving chip/circuit power efficiency is worth close study. In this context, approximate computing recently arises as new research trend [1–6]. In conventional designs, datapath computations are precise for its bitwidth, i.e., the error is restricted to those less than the least significant bit weight. We term them as *per-bitwidth precise computing*. Approximate computing exploits intrinsic error tolerance in certain applications, such as image processing and machine learning, and allow occasional small errors beyond the quantization error caused by limited bitwidth. By doing so, circuits can be implemented in a less power hungry fashion compared to per-bitwidth precise circuits. The research on approximate computing so far encompasses approximate arithmetic circuit design [1, 2, 4], logic synthesis [3], modeling [7–10], specific applications [6] and architectural level techniques.

Besides power, the growing chip complexity also becomes a challenge and increasingly necessitates the use of automated system level design or high level synthesis (HLS). High level decisions usually generate large impact to the overall system performance and power. Given a rich body of low/circuit level approximation techniques, how to efficiently utilize them at system level is of great importance but not well studied yet. In [5], an automated design space exploration technique is introduced to choose between approximate and per-bitwidth precise implementations for each operation. However, it pays little attention to scheduling and resource allocation/binding algorithms, which are the core techniques in HLS.

This work attempts to find techniques that make efficient use of approximate circuits at system/block level by considering them in HLS. While conventional HLS mostly emphasizes performance and power/area, the notion of approximate computing requires error control in addition. Existing error models for approximate computing [7–10] are mostly targeted to post-design analysis and very difficult to be used within optimizations. In this work, we first study optimization-friendly error models. Then, we propose and investigate two HLS approaches that cover scheduling, Functional Unit (FU) allocation/binding together with approximation assignment. We focus on data-intensive applications as opposed to control-intensive applications since approximate computing mostly occurs at datapaths.

The error control in approximate computing is akin to bitwidth optimization [11, 12] where bitwidth of each operation is selected to minimize implementation cost subject to precision constraints. Bitwidth optimization has also been studied together with HLS [13, 14] where the HLS and bitwidth optimization interact with each other but are carried out separately. In per-bitwidth precise designs, the quantization error analysis must be accurate and thus is too complex to be incorporated within HLS. By contrast, the requirement for accuracy is less strict in approximate computing. Thus, we can integrate bitwidth optimization and approximation assignment with HLS such that the solution space is searched more thoroughly.

The proposed techniques are evaluated by simulations on benchmark applications including Verilog simulations. The results confirm the effectiveness of our techniques. To our knowledge, this is the first work on scheduling and resource allocation/binding for approximate computing systems. The contributions of this paper include:

DAC '15 June 07 - 11, 2015, San Francisco, CA, USA

ACM 978-1-4503-3520-1/15/06$15.00
http://dx.doi.org/10.1145/2744769.2744863

- We propose a variance-based error model that is very simple to use in optimizations. It is enhanced by error sensitivity to capture structural correlations in error propagation. Its credibility is validated by Verilog-based Monte Carlo simulations.
- A multiple choice multiple dimension Knapsack formulation is introduced for precision optimization that concurrently considers approximation selection and bitwidth optimization.
- An iterative list scheduling heuristic is developed to perform simultaneous scheduling, FU allocation/binding with consideration of approximation.
- An ILP (Integer Linear Programming) formulation is presented for integrated precision optimization and HLS.

2. FORMULATION AND NOTATIONS

The input to our algorithms is a task graph or dataflow graph $G(V, E)$ where the node set V is composed by disjoint subsets of primary inputs PI, computation operation nodes $\hat{V}$ and primary outputs PO, and the edges E indicate data dependencies. In this work, we focus on topology of directed acyclic graph. Each node $\omega \in \hat{V}$ has an operation type τ_ω, such as addition and multiplication.

In approximate computing, a key part is precision control. When there are multiple approximation options, e.g., the approximate adder [1] can be implemented with different numbers of imprecise bits, we need to decide how to choose among different approximate implementations. Further, approximate implementation can be considered together with bitwidth optimization [11], which decides how many bits are utilized in implementing a computation. The conventional bitwidth optimization [11] consists of two parts: one that decides the data range and the other that affects the computing precision. We focus on the precision part as it coheres better with approximate computing.

For each operation type τ, there is a set of implementations $\mathcal{I}_\tau = \{F_\tau^{\phi_1}, F_\tau^{\phi_2}, ...\}$ of different precisions $\phi_1, \phi_2, ...$ and we use $\mathcal{I}_\tau^{\succeq \phi}$ to represent the subset in $\mathcal{I}_\tau$ that has at least precision ϕ. The precision optimization is to find the lowest precision level $\underline{\phi}(\omega)$ for each $\omega \in \hat{V}$ such that the system precision specification is satisfied.

The HLS in this work considers FU allocation, binding and scheduling. Binding is to associate an operation $\omega \in \hat{V}$ with an FU instance f of implementation $F \in \mathcal{I}_{\tau_\omega}$ and we also use the notation $F(\omega)$ and $f(\omega)$ to tell the implementation and FU instance for ω, respectively. We allow operation ω to be bound to an FU of precision higher than $\underline{\phi}(\omega)$ in order to encourage FU sharing, i.e., ω can be bound to any instance of $F \in \mathcal{I}_{\tau_\omega}^{\succeq \underline{\phi}(\omega)}$. Given a latency constraint Q, the scheduling is to find start time $0 \leqslant s(\omega) < Q - \Lambda_{F(\omega)}, \forall \omega \in \hat{V}$, where $\Lambda_{F(\omega)}$ is the execution latency for the FU of ω. When we try to bind/schedule an operation but there is no corresponding FU available, a new FU is allocated. Thus, the allocation is decided along with binding and scheduling. The overall objective is to minimize total leakage energy consumption while latency and system precision constraints are satisfied. The extension to include dynamic energy is not difficult. Leakage energy is also highly correlated with FU cost, which is a typical objective function in conventional HLS.

3. ANALYTICAL ERROR MODELS

If not carefully used, approximation may result in errors that are extravagant and beyond acceptable level. Hence, precision control and error models are of critical importance for approximate computing. An error model should quantify the difference between the precision of a system implementation and the precision specification. If a model is to be employed within optimization algorithms, i.e., to guide each solution search step during optimization, it must be simple to compute as it would be called very frequently. Meanwhile, it must be credible and close to accurate analysis.

The work of [8] attempts to find a couple of error models that do not rely on time consuming simulations. One is based on interval arithmetic, which can be overly pessimistic, and the other is based on affine arithmetic, which has poor storage scalability. Moreover, both techniques entail lookup tables in error propagation, which is restrictive to use in optimizations. Error rate [10], which is the probability that an approximate result is different from its precise counterpart, is simple to use by taking logarithm. However, it only addresses error frequency without attention to error magnitude. A variance based error model is described in [12]. However, it neglects structural correlations among signal propagations, which can be quite remarkable.

We now discuss how errors are propagated in a couple of typical operations. Consider addition operation $y = a + b$ and let errors of a, b, the addition and s be denoted by ϵ_a, ϵ_b, ϵ_+ and ϵ_y, respectively. Then, the approximate addition can be represented by

$$y + \epsilon_y = (a + \epsilon_a) + (b + \epsilon_b) + \epsilon_+. \tag{1}$$

Likewise, the error propagation in multiplication $p = a \times b$ is described by

$$p + \epsilon_p = a \cdot b + a\epsilon_b + b\epsilon_a + \epsilon_\times \cancel{+ \epsilon_a\epsilon_b}. \tag{2}$$

For the sake of simplicity without significant loss of accuracy, we neglect the second order error $\epsilon_a\epsilon_b$.

We propose an error model where each error ϵ is treated as a random variable. Then, an error can be characterized by its mean $\mu(\epsilon)$ and variance $\nu(\epsilon)$. We argue that the mean error $\mu(\epsilon)$ at a system/block output under the operations of approximate computing is systematic and can be compensated by a constant offset. Then, the overall computation precision is determined by the variance $\nu(\epsilon)$.

Lemma: *If an error ϵ is a random variable, its variance after the constant compensation is equal to the Mean Squared Error (MSE)*[1].

MSE is a common error model in approximate computing [10] and equivalent to Peak Signal Noise Ratio, which is employed in [1, 2]. According to Equations (1) and (2), the error of system output is a linear combination of the operation errors. The variance of a linear combination of random variables $X_1, X_2, ..., X_n$ is

$$\nu(\sum_{i=1}^{n} k_i X_i) = \sum_{i=1}^{n} k_i^2 \nu(X_i) + 2\cancel{\sum_{i=1}^{n} \sum_{j=i+1}^{n} k_i k_j cov(X_i, X_j)}, \tag{3}$$

[1] The proof is simple and omitted due to space limit.

where k_i denotes constant coefficient and $cov(X_i, X_j)$ is the covariance between X_i and X_j.

The overall system error is a composite effect due to error generation, like the ϵ_+ in Equation (1) and $\epsilon_\times$ in Equation (2), and error propagation like ϵ_a and ϵ_b in Equation (1) and (2). The error generations among different operations are largely independent of each other, except a rare case where two operations use the same implementation and the same input data. Therefore, we drop the covariance term in our model. This simplification also helps to avoid nonlinear terms of decision variables in optimization.

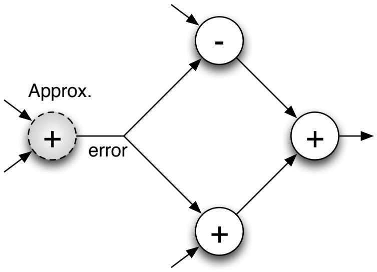


Figure 1: An error from an approximate adder may cancel itself along reconvergent paths.

Error propagations exhibit strong structural correlations that cannot be ignored. For example, an error from the approximate adder in Figure 1 is propagated along two paths and it is subtracted in the upper path. When the two paths reconverge, the error from the lower path is canceled by the one propagated along the upper path. We propose the concept of *error sensitivity* (ES) to capture the first order effect of such structural correlation. For a single error ϵ_ω from operation $\omega \in \hat{V}$ and the error $\epsilon_{\omega,o}$ incurred by ϵ_ω at a primary output $o \in PO$, the error sensitivity is defined as $ES_{\omega,o} = \frac{\epsilon_{\omega,o}}{\epsilon_\omega}$. Then, the error variance of an output $o \in PO$ can be expressed as

$$\nu(\epsilon_o) = \sum_{\forall \omega \in \hat{V}} ES^2_{\omega,o} \cdot \nu(\epsilon_\omega). \tag{4}$$

Please note that ES is squared here like the coefficient k in Equation (3).

The ES of each node $\omega \in \hat{V}$ can be obtained through an extension to depth first search (DFS) of G. If all operations are addition, $\epsilon_{\omega,o}$ is simply $n \times \epsilon_\omega$, where n is the number of distinct paths from node ω to output o, and thus $ES_{\omega,o}$ is n. If the error ϵ_ω experiences a scaling $\times K$ operation along one of the paths to o, $ES_{\omega,o}$ is $(n-1)+K$. If it is multiplied by another variable, the error propagation is approximated by scaling of an empirical value. Given a task graph, the DFS-based ES estimation is performed once as a pre-processing.

4. KILS: KNAPSACK AND ITERATIVE LIST SCHEDULING

We describe a sequential heuristic that first decides precisions, considering both approximation selection and bitwidth optimization, and then performs a list scheduling-based HLS algorithm with consideration of approximation.

4.1 Knapsack-Based Precision Optimization

In the simplest case, each operation $\omega \in \hat{V}$ has only one approximate implementation, i.e., $|\mathcal{I}_{\tau_\omega}| = 1$. A binary decision variable $x_\omega \in \{0, 1\}$ tells if to choose approximation for ω or not. If the energy saving for using approximation at ω is Ψ_ω, we wish to maximize the total energy savings subject to error variance constraint at the output. If there is a single output at G, the formulation is

$$\text{maximize} \sum_{\omega \in \hat{V}} \Psi_\omega \cdot x_\omega \tag{5}$$

$$\text{s.t.} \sum_{\omega \in \hat{V}} ES^2_\omega \cdot \nu(\epsilon_\omega) \cdot x_\omega \leqslant \nu_B \tag{6}$$

where ν_B is the error variance bound at the output. This formulation is the well-known 0-1 knapsack problem. Note that inequality (6) has a linear form as a result of removing the covariance term in Equation (3).

When we consider multiple approximation implementations, simultaneous bitwidth optimizations and multiple output nodes, the above formulations can be extended as a Multiple choice Multiple dimension Knapsack Problem (MMKP). In MMKP, a set of items are partitioned into n classes and k dimensions. One needs to choose exactly one item from each class such that the overall benefit is maximized while the capacity constraint of each dimension is satisfied. To our case, each operation $\omega \in \hat{V}$ is a class, which corresponds to a set $\mathcal{I}_{\tau_\omega}$ of implementations, and each dimension is for one output $o \in PO$ with error variance bound ν_{B_o}. The decision variable becomes $x_{\omega,F} \in \{0, 1\}$, which tells if to choose $F \in \mathcal{I}_{\tau_\omega}$ for operation $\omega \in \hat{V}$. The formulation is:

$$\text{maximize} \sum_{\omega \in \hat{V}} \sum_{F \in \mathcal{I}_{\tau_\omega}} \Psi_{\omega,F} \cdot x_{\omega,F}$$

$$\text{s.t.} \sum_{\omega \in \hat{V}} \sum_{F \in \mathcal{I}_{\tau_\omega}} ES^2_{\omega,o} \cdot \nu(\epsilon_{\omega,F}) \cdot x_{\omega,F} \leqslant \nu_{B_o}, \quad \forall o \in PO$$

$$\sum_{F \in \mathcal{I}_{\tau_\omega}} x_{\omega,F} = 1, \quad \forall \omega \in \hat{V}$$

Please note the set $\mathcal{I}_{\tau_\omega}$ includes implementations of different bitwidths, different approximation and their combinations for operation type τ_ω. As such, bitwidth optimization and approximation selection are carried out in an integrated manner. We solve this MMKP problem using an ILP solver. The result tells the *lowest precision* $\underline{\phi}(\omega)$ for each node $\omega \in \hat{V}$ such that the overall system precision specification is satisfied. Please note the $\underline{\phi}(\omega)$ is not a commitment but a guidance to the subsequent HLS.

4.2 Approximation-Aware HLS

4.2.1 Conventional List Scheduling

One popular heuristic for HLS is the list scheduling [13], which has two variants: one is to minimize resource cost subject to latency constraint and the other is to minimize latency subject to resource constraint. We take the former variant as a basis due to its similarity to our problem formulation, and make a remarkable extension to take approximation/precision into account. The list scheduling handles FU allocation/binding as well.

The core part of conventional list scheduling is preceded by ASAP (As Soon As Possible) and ALAP (As Late As Possible) schedulings. For each node $\omega \in \hat{V}$, the lower (upper) bound of its start time $\underline{t}_\omega$ ($\bar{t}_\omega$) is its ASAP (ALAP) schedule. Then, the range of its schedule is initially $[\underline{t}_\omega, \bar{t}_\omega]$. Next, the core algorithm is a one-pass topological order traversal of the task graph G. During the traversal, a ready list maintains the nodes whose precedent nodes are either in PI or have already been scheduled. Among all nodes in the ready list, the one $\omega \in \hat{V}$ with the minimum $\bar{t}_\omega$ is selected to be scheduled. If an FU f that implements ω has already been allocated and is available in $[\underline{t}_\omega, \bar{t}_\omega]$, then ω is bound to f and scheduled to the earliest available time of f in $[\underline{t}_\omega, \bar{t}_\omega]$. If no such FU can be found, a new FU is allocated and bound to ω. This procedure is repeated till all nodes are scheduled.

4.2.2 Problems of Conventional List Scheduling

When approximation/precision is considered, a binding has multiple implementation options of asymmetric compatibility. That is, a low precision operation can be bound to a high precision FU of the same type, but not vice versa. This creates a subtlety that makes the conventional list scheduling inefficient. Consider the example in Figure 2 where an adder delay is 1 and a multiplier delay is 2. In conventional list scheduling, the approximate multiplication is first bound to an approximate multiplier and later a precise multiplier must be allocated and bound to the precise multiplication, as shown in Figure 2 (b). However, the two multiplications can share a single precise multiplier as in Figure 2 (c).

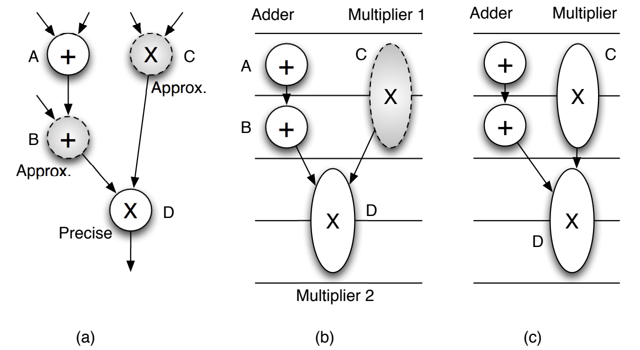


Figure 2: (a)Task graph; (b) Conventional list scheduling results in two multipliers; (c) The two multiplications can share one multiplier.

Even if approximation is not considered, the conventional list scheduling may be inefficient due to its myopic nature. This is illustrated by the example in Figure 3, where the latency deadline is 3 and an adder delay is 1. In the conventional scheduling, when node B is considered for scheduling, it can be bound to $Adder1$ without violating latency constraint. Then, node C can no longer use $Adder1$ due to the latency constraint and $Adder2$ is allocated. In the last time step, all of nodes D, E and F must be scheduled to satisfy the latency constraint. Overall, three adders are allocated. However, one can see that actually only two adders are necessary, as shown in Figure 3 (c).

4.2.3 New Iterative List Scheduling

In order to solve the aforementioned problems, we propose two significant changes to the list scheduling and show the overall pseudo code in Algorithm 1. We first change the algorithm to be iterative instead of one-pass graph traversal.

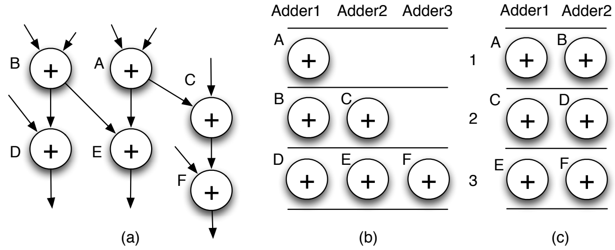


Figure 3: (a)Task graph; (b) Conventional list scheduling results in 3 adders for latency constraint of 3; (c) Two adders are sufficient.

The iterations are shown as the **while** loop in step 7, where $\mathcal{F}^i$ indicates the set of FUs allocated in iteration i. The first iteration generates the initial FU allocation and the subsequent iterations try to reduce the FU allocation.

```
1  Approximation_Aware_HLS(G(V,E)) begin
2      foreach $\omega \in \hat{V}$ do
3          $\underline{t}_\omega \leftarrow ASAP\_schedule$, $\bar{t}_\omega \leftarrow ALAP\_schedule$
4          $\mathcal{F}^0_{\tau_\omega} \leftarrow \emptyset$ // Initialize allocated FUs for $\tau_\omega$
5      end
6      $i = 0$
7      while $i \leqslant 1$ or $|\mathcal{F}^i| < |\mathcal{F}^{i-1}|$ do
8          $i++$
9          foreach $\omega \in \hat{V}$ do
10             $\mathcal{F}^i_{\tau_\omega} \leftarrow \emptyset$
11         end
12         Initialize $Ready\_List$ from PI
13         while $Ready\_List \neq \emptyset$ do
14             $\omega \leftarrow$ node with min $\bar{t}$ in $Ready\_List$
15             $Candidates^{i-1} \leftarrow f \in \mathcal{F}^{\succeq\phi(\omega),i-1}_{\tau_\omega}$ & free in $[\underline{t}_\omega, \bar{t}_\omega]$
16             $Candidates^{i} \leftarrow f \in \mathcal{F}^{\succeq\phi(\omega),i}_{\tau_\omega}$ & free in $[\underline{t}_\omega, \bar{t}_\omega]$
17             if $Candidates^{i-1} \neq \emptyset$ or $Candidates^{i} \neq \emptyset$ then
18                 if $\Upsilon^{Candidates^{i-1}}_{\prec\omega} < \gamma \cdot \Upsilon^{Candidates^{i-1}}$ then
19                     Find $\hat{f} \in Candidates^{i-1}$ with min start time
20                     break tie with max $\Upsilon$
21                 else
22                     Find $\hat{f} \in Candidates^{i}$ with max $\Upsilon$
23                     break tie with min start time
24                 end
25             else
26                 Allocate $\hat{f}$ of $F^{\phi(\omega)}_{\tau_\omega}$
27             end
28             $\mathcal{F}^i_{\tau_\omega} \leftarrow \mathcal{F}^i_{\tau_\omega} \cup \{\hat{f}\}$,   $f(\omega) \leftarrow \hat{f}$ // binding
29             $s(\omega) \leftarrow$ earliest available time for $\hat{f}$ in $[\underline{t}_\omega, \bar{t}_\omega]$
30             Update $Ready\_List$, update $[\underline{t}_\omega, \bar{t}_\omega], \forall\omega \in \hat{V}$
31         end
32     end
33 end
```

Algorithm 1: Algorithm for approximation aware HLS.

The second and more important change is step 15-24. In step 15 and 16, a set of candidate FUs are identified from the FUs $\mathcal{F}^{\succeq\phi(\omega)}_{\tau_\omega}$ that are allocated in current and the previous iterations. The "free in $[\underline{t}_\omega, \bar{t}_\omega]$" means that the FU is available from a time in $[\underline{t}_\omega, \bar{t}_\omega]$ to an extent long enough to accommodate one complete operation on the FU. The selection of FUs among the candidates is based on two factors. (i) Early start time: if an operation is scheduled to start early, greater slack (or mobility) is left to subsequent nodes, which consequently have greater chance to share FUs and reduce the number of FUs. (ii) Utilization: if we move operations from FUs with low utilization to those with high utilization, there would be greater chance to empty some FUs.

We define *utilization* $\Upsilon^{\mathcal{F}}$ of a set $\mathcal{F}$ of FUs as the ratio of the total time the FUs are used versus the total latency multiplied by the number FUs in the set. For example, the utilization of *Adder*3 in Figure 3 (b) is $\frac{1}{3}$ and the utilization for all three adders is $\frac{6}{3\times 3}$. We also define *ancestor utilization* $\Upsilon^{\mathcal{F}}_{\prec\omega}$ with respect to operation $\omega \in \hat{V}$ as the ratio of the total time the FUs in $\mathcal{F}$ being used by ancestor nodes of ω versus the wall-to-wall time of these uses multiplied by the number of FUs in $\mathcal{F}$.

Ideally, we wish all FUs are fully and equally loaded by operations. In other words, at any specific time step, the FU utilization is preferred to be equal to the overall utilization among all FUs. In step 18, we choose the utilization for all candidates $\Upsilon^{Candidates^{i-1}}$ scaled by a correction factor γ as the target. If the utilization of candidate FUs by ancestor nodes of ω is less than the target, the utilization so far is relatively low and we prefer to schedule ω as early as possible. Otherwise (step 22), we only use FUs allocated in current iteration and prefers to squeeze the operation into the FU with high utilization. Steps 20 and 23 tell how break tie.

Now let us see how our algorithm solves the case of Figure 2. After the first iteration, as in Figure 2 (b), one approximate multiplier and one precise multiplier are allocated. In the second iteration, when node C is considered, both multipliers are candidates and the corresponding $\Upsilon^{Candidates^1}$ is $\frac{1}{2}$. As there is no node preceding C, we go to branch of step 19. Since both multipliers can be scheduled to time step 1, we break tie according to utilization. Excluding operation C, the utilization of the approximate multiplier is 0 while the utilization of the precise multiplier is $\frac{1}{2}$. Thus, node C is bound to the precise multiplier and the approximate multiplier is no longer used in the second iteration.

For the example in Figure 3, we reach (b) after the first iteration. When we try to schedule node B in the second iteration, the $\Upsilon^{Candidates^1}_{\prec B}$ means the utilization of *Adder*1 in time step 1, which is $\frac{1}{3}$. This is less than the overall utilization of $\frac{2}{3}$ and therefore we go to step 19 to bind B with *Adder*2 and start B at time step 1. Eventually, the second iteration would reach a result like Figure 3 (c).

5. INTEGER LINEAR PROGRAMMING

The precision optimization, scheduling, FU allocation and binding can be performed simultaneously through ILP:

$$\text{Min} \quad \sum_F L_F \cdot u_F \cdot Q \tag{7}$$

$$\text{s.t.} \quad \sum_{0\leqslant t<Q} \sum_F x_{\omega,t,F} = 1,\ \forall \omega \in \hat{V} \tag{8}$$

$$\sum_t \sum_F x_{\omega,t,F} \cdot t - s(\omega) = 0,\ \forall \omega \in \hat{V} \tag{9}$$

$$s(v) + \sum_t \sum_F x_{v,t,F} \cdot \Lambda_F \leqslant s(\omega),\ (v,\omega) \in E \tag{10}$$

$$s(o) \leqslant Q,\ \forall o \in PO \tag{11}$$

$$\sum_{t|t\leqslant\hat{t}<t+\Lambda_F} \sum_{\omega\in\hat{V}} x_{\omega,t,F} \leqslant u_F \leqslant U_F, 0 \leqslant \hat{t} < Q, \forall F \tag{12}$$

$$\sum_{\omega\in\hat{V}} \sum_t \sum_F ES^2_{\omega,o}\nu(\epsilon_F)x_{\omega,t,F} \leqslant \nu_{B_o}, \forall o \in PO \tag{13}$$

The latency constraint is denoted as Q and thus all operations must be executed in the time range $0 \leqslant t < Q$. Decision variable $x_{\omega,t,F} \in \{0,1\}$ tells if operation $\omega \in \hat{V}$ is scheduled to start at time t and bound to an FU of implementation F. The leakage power and latency of implementation F are represented by L_F and Λ_F, respectively. The number of FUs of implementation F being allocated is u_F and U_F is a constant upper bound for u_F. The start time of operation ω is denoted by $s(\omega)$.

The objective here is to minimize the total leakage energy. Constraint (8) ensures that each operation is scheduled to only one time and bound to only one FU. Inequality (10) is the precedence constraint and inequality (11) is the latency constraints at PO. The FU allocation is realized through inequality (12). The last constraint is to bound variance at each PO node.

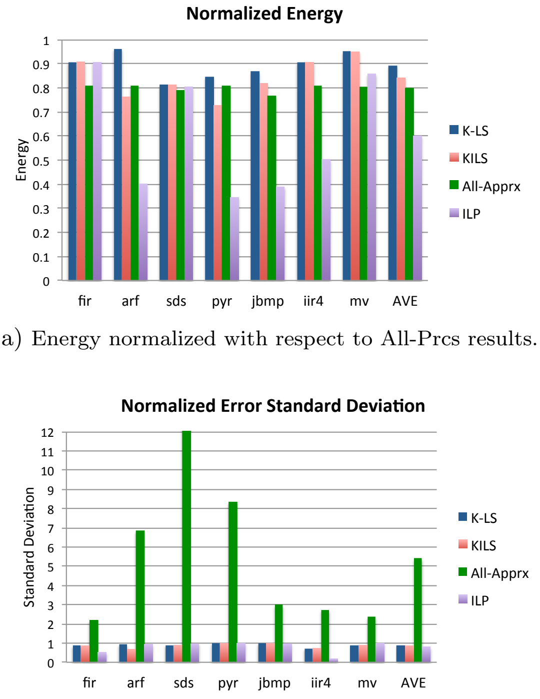
6. EXPERIMENT

We implemented gate level designs of approximate adder [1] and approximate multiplier [4], along with 24-bit, 28-bit and 32-bit precise adder/multipliers using $15nm$ technology. Energy and latency are characterized through SPICE simulations while error variance is obtained through Verilog-based Monte Carlo simulations. The experiments are performed on a set of MediaBench applications [15] and some other common applications like FIR, IIR and ARF, each of which has about 10 to 100 nodes. The ILP and Knapsack problems are solved by the CPLEX Optimizer [16]. Since there is no previous work on scheduling/binding considering approximate computing, we compare the following methods:

- All-Prcs: all FUs take the most precise implementation upon the conventional list scheduling result.
- All-Apprx: all FUs choose an approximate implementation upon the conventional list scheduling result.
- K-LS: our Knapsack based precision optimization followed by conventional list scheduling.
- KILS: our Knapsack based precision optimization followed by our iterative list scheduling.
- ILP: our integer linear programming approach.

The main results are shown in Figure 4, where (a) is for energy and (b) is for error standard deviation, which is equivalent to variance. Unlike variance that is in the dimension of square of data, standard deviation has the same dimension as data and provides more intuitive sense. The rightmost clusters of bars are the average results. All results satisfy latency constraints. The energy savings from K-LS, which is the conventional list scheduling, is about 11% while our ILP can achieve average energy reduction of 40%. Our KILS heuristic also outperforms the conventional approach by reducing 16% energy. ILP often results in less energy than All-Apprx as it uses less number of FUs. Figure 4 (b) shows that all methods except All-Apprx satisfy the error constraint. Without error control, the All-Apprx method causes standard deviation about 5× of the constraints, although it provides 20% energy savings.

In order to confirm the credibility of our variance-based model, we implement the ILP results in Verilog for two cases - ARF (Auto Regression Filter) and FIR filter. We run Verilog-based 20K-run Monte Caro simulations to obtain MSE at the outputs, which are compensated by constant offsets to nullify the mean errors. We observe that there is about 10% difference between our variance and the



(a) Energy normalized with respect to All-Prcs results.

(b) Error standard deviation normalized with respect to standard deviation constraints.

Figure 4: Results from MediaBench applications.

simulated MSE. However, the *correlation coeffcient* between them is 0.94, which means they are highly correlated. We further plot the energy-error tradeoff curves using the two different error models in Figure 5. For each of FIR and AFR cases, the curves from the two models are almost identical. This reminds the Elmore delay model, which is inaccurate but has high fidelity and provides reliable guidance in optimizations. Figure 5 also confirms that our approach can realize different energy-error tradeoffs.

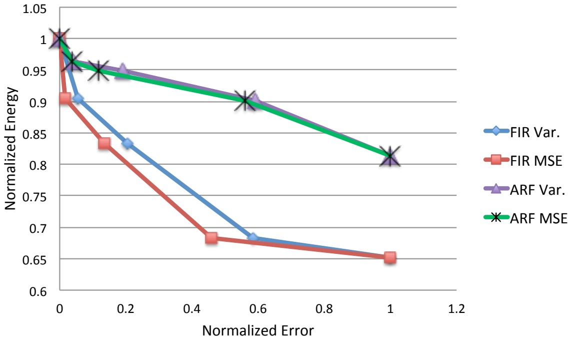


Figure 5: Energy-error tradeoff from ILP result. The MSE results are from Verilog simulations.

We compare runtime of KILS and ILP in Figure 6. One can see that KILS is usually one order of magnitude faster than ILP. Moreover, KILS has better scalability. KILS has an advantage in handling large cases, although its solutions are not as good as those from ILP.

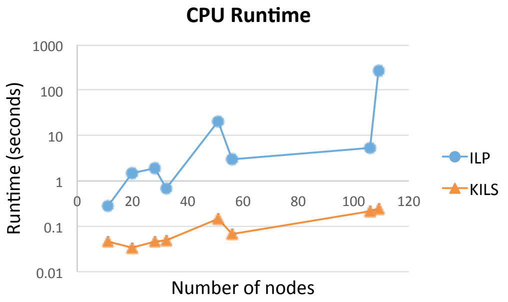


Figure 6: Runtime (in log scale) comparison.

7. CONCLUSIONS AND FUTURE WORK

To our knowledge, this is the first work on scheduling/binding algorithms considering approximate circuits. A simple yet credible error model is proposed. We develop a sequential heuristic and an ILP based exact approach for joint precision optimization and approximation-aware HLS. The approaches provide energy vs. error tradeoff at system level. In future research, we will consider interconnect/mux and register binding. We will also investigate cases which are both data-intensive and control-intensive.

8. REFERENCES

[1] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. IMPACT: imprecise adders for low-power approximate computing. *ISLPED*, pages 409–414, 2011.

[2] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. *ICCAD*, pages 728–735, 2012.

[3] J. Miao He, A. Gerstlauer, and M. Orshansky. Approximate logic synthesis under general error magnitude and frequency constraints. *ICCAD*, pages 779–786, 2013.

[4] C. Liu, J. Han, and F. Lombardi. A low-power, high-performance approximate multiplier with configurable partial error recovery. *DATE*, 2014.

[5] K. Nepal, Y. Li, R. I. Bahar, and S. Reda. ABACUS: a technique for automated behavioral synthesis of approximate computing. *DATE*, 2014.

[6] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. AxNN: energy-efficient neuromorphic systems using approximate computing. *ISLPED*, pages 27–32, 2014.

[7] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: modeling and analysis of circuits for approximate computing. *ICCAD*, pages 667–673, 2011.

[8] J. Huang, J. Lach, and G. Robins. Analytic error modeling for imprecise arithmetic circuits. *SELSE*, 2011.

[9] J. Liang, J. Han, and F. Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Trans. on Computers*, 62(9):1760–1771, June 2012.

[10] W.-T. J. Chan, A. B. Kahng, S. Kang, R. Kumar, and J. Sartori. Statistical analysis and modeling for error composition in approximate computation circuits. *ICCD*, pages 47–53, 2013.

[11] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE TCAD*, 25(10):1990–2000, October 2006.

[12] S. Lee and A. Gerstlauer. Fine graind word length optimization for dynamic precision scaling in DSP systems. *VLSI-SoC*, pages 266–271, 2013.

[13] K.-I. Kum and W. Sung. Combined word-length optimization and high-level synthesis of digital signal processing systems. *IEEE TCAD*, 20(8):921–930, August 2001.

[14] J. Cong, Y. Fan, G. Han, Y. Lin, J. Xu, Z. Zhang, and X. Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. *ASPDAC*, pages 856–861, 2005.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. *MICRO*, pages 330–335, 1997.

[16] IBM. CPLEX Optimizer. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/, 2014.