Approximation on Data Flow Graph Execution for Energy Efficiency

Qian Xu, Md Tanvir Arafin, Gang Qu

Abstract Data flow graph (DFG) is a popular model for software and its execution. It consists of a list of arithmetic operations without conditionals and their dependencies. Completion time and energy consumption are two main objectives for DFG optimization. In this chapter, we discuss approximation methods at different levels of DFG that can reduce energy consumption with a guaranteed quantity of results. First, we consider a probabilistic design framework that approximates the application by intentionally terminating certain DFG executions before reaching the deadline. Second, we demonstrate a real-time estimation-and-recomputing approach that executes the non-critical parts of the DFG with approximation. Finally, we use the floating-point logarithmic operation as an example to show how to optimize data bit width based on the DFG model.

1 Introduction

Advances in hardware design methodology and semiconductor fabrication technology continue to enable the rapid growth of computer systems with high computation speed and low power consumption, despite the fact that we are getting to the end of Moore's Law that predicts the shrinking of transistor size and the increasing of the number of transistors per chip area. Meanwhile, with the explosion of data and the prevalence of the Internet, there is still a high demand to produce low-power, high-speed end devices, especially for the Internet of Things (IoT). This desire inspires plenty of research focusing on exploiting approximation to improve the system's

Qian Xu
University of Maryland, College Park, Maryland e-mail: qxu1234@umd.edu

Md Tanvir Arafin
Morgan State University, Baltimore, Maryland e-mail: mdtanvir.arafin@morgan.edu

Gang Qu
University of Maryland, College Park, Maryland e-mail: gangqu@umd.edu

overall performance by balancing the amount of computation, which determines power and energy, and the quality of the computation, which can be evaluated by metrics such as result accuracy [16].

It should be noted that approximation is not suitable for all applications or hardware systems. For example, when a system requires high precision and energy is not the primary concern, accuracy should not be traded for power and energy. Therefore, most of the reported approximate computing approaches are mainly designed for applications such as multimedia processing and machine learning. The former relates to the human cognition system, which accepts (and, to some extent, cannot detect) minor errors. The latter are examples of error-resilient systems where small computational errors will not impact the outcome. In short, before applying approximation methods to a system, one needs to consider whether the specific system can tolerate the errors introduced by approximation.

As a basic building block of any computer system, hardware components that perform arithmetic operations are the foundation of computation. Their high occurrence in a system makes them one of the most popular research topics in approximate computing. Various types of approximate adders [12], approximate multipliers [17, 15], and approximate dividers [7] have been proposed and demonstrated to be effective in reducing power and energy consumption. Different data formats, such as integers, fixed-point numbers, floating-point numbers, and new data formats [5] have also been explored for approximate computation.

Data flow graph (DFG) is a popular model for computation at various abstract levels, from the system level and application level to individual operations such as addition, subtraction, and multiplication. It has been extensively used in the low power implementation of software [10, 11, 14]. In a DFG, the nodes represent computations, the edges represent the flow of data or data dependencies between the computations, and the inputs to the DFG are variables or constants. The aforementioned approximation approaches have considered the input data (approximation at data level) and the nodes (approximation at basic arithmetic units) of a DFG for optimization. Such local optimization methods fail to utilize the synergy among the data and computation in a DFG. Many applications and programs require many iterations of the same DFG. Therefore, we believe they will not achieve approximate computing's full potential in reducing power and energy. In this chapter, we demonstrate this through several projects.

2 Chapter Overview

This chapter aims to provide a comprehensive view of the approximate computing on DFG execution for energy efficiency. This will be quite different from the existing approximation approaches on single computation units or input data. We will consider the entire DFG or the underlying application/program represented by the DFG.

In Section 3, we first describe the main structure and components of DFGs. We then introduce the three main research questions for any DFG approximation scheme. We summarize the key ideas of the three DFG approximation algorithms that will be elaborated on in the remainder of this chapter.

In Section 4, we consider the iterative execution nature of the DFGs, such as those used in multimedia applications, and present a probabilistic design method to implement DFGs [10, 11]. Unlike the traditional approach that attempts to complete each iteration of the DFG execution, we intentionally terminate certain iterations before they miss the execution deadline to save energy. Such early termination may not guarantee the maximal number of completed iterations of the DFG. Still, it can provide us significant energy savings if the program does not require the highest completion ratio. In short, this method trades the program's completion ratio for energy efficiency.

In Section 5, we discuss another approximation technique on the single execution of a DFG. We identify the critical and non-critical branches in the DFG and perform accurate and approximate computing on them, respectively, to reduce energy consumption without large compromise to the precision of the computation. This approach is referred to as *estimate and recompute* [4].

In Section 6, we consider how approximation inside the operations can affect the overall performance in DFGs and discuss the significance of accommodating specific processes in the whole program. Using the logarithmic function as an example, we show how to decide the approximation level for each operation so that the best performance can be achieved through the cooperation of multiple processes [28].

Section 7 summarizes this chapter of discussion on approximation on DFG execution for energy efficiency.

3 Approximation on DFG Execution: Challenges and Solutions

3.1 Data Flow Graph Basics

A Data Flow Graph (DFG) is a directed acyclic graph where each node represents a computation (such as addition and multiplication), and an edge from node u to node v indicates that the computation result of node u will be an input for the computation at node v. Edges without a source node are input to the DFG and edges without destination node are outputs of the DFG. For example, consider a basic algebraic operation, the dot product, of two 1-D two-element vectors $a = [a_0, a_1]$ and $b = [b_0, b_1]$. Let c be the result of the dot product, then c can be calculated following Equation 1.

$$c = a_0 * b_0 + a_1 * b_1 \tag{1}$$

In Equation 1, there are two multiplication operators and one addition operator. If we denote the first product by t_0 and the second product by t_1. It is easy to obtain the data dependencies that the addition at node c depends on the two intermediate

products t_0 and t_1 while the two multiplications at nodes t_0 and t_1 depend on the availability of their operands, (a_0,b_0) and (a_1, b_1). Figure 1 is the DFG constructed based on Equation 1, the dot product.

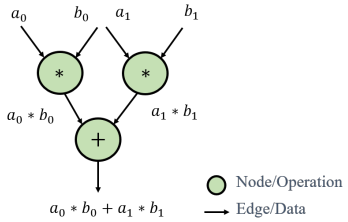


Fig. 1 DFG example of dot product on two 1-D two-element vectors.

For any sequence of operations without conditionals, a DFG can be generated. Addition and multiplication, division, and even some complex operators like square root or logarithm can be represented as nodes. The numbers in all formats, including integer, fixed-point, floating-point, and even user-defined format, can be represented as edges. As discussed earlier, approximate computing on DFGs has been done on both the nodes (in the form of an approximate hardware implementation of the operations) and edges (on data approximation with different formats). But there is little work on how to consider the DFG systematically for approximation. In this chapter, we discuss how to fill this gap with approximate computing techniques focusing on the execution of the DFG instead of its nodes or edges.

3.2 Main Questions for Approximation on DFGs and Their Execution

Among the many considerations, answers to the following three questions are crucial for developing any approximate computing method for an application modeled as a DFG and its execution.

- **What is approximable:** whether the entire application is error-resilient or which portions can tolerate errors? The answer to this question determines the targets for approximation.
- **How to approximate the designs:** which specific approximation techniques are applicable for the given application? The answer to this question focuses on the details of approximation.
- **How to ensure the output quality:** to which extend can we approximate the application and its execution? The answer to this question will be application-specific, and an optimal solution will balance the trade-off between energy efficiency and output quality.

Discovering the approximable portions in a DFG is not easy. First of all, this is application-specific, and a well-understanding of the DFG-represented application

and its execution characteristics is necessary. At a high level, when we consider the entire DFG, applications like multimedia signal processing are approximable. Take video decoding, for example, failing to decode one frame will most likely be acceptable. At a low level, when we consider the nodes and edges in the DFG, there are many opportunities for approximation. One can use approximate adders and multipliers as the basic building blocks of DFGs to approximate the corresponding operations. One can also choose to approximate the input data to the DFG or certain nodes of the DFG to reduce the amount of computation and achieve energy efficiency.

After the approximation targets in the DFG are determined, there are still many different approximation techniques to choose from. For example, more than a dozen approximate adders and approximate multipliers have been proposed in the past. As a result, the options for approximation on a DFG will be exponential to the number of operations units such as adder and multiplier that the DFG has. As another example, the impact of each node on the output's precision may not be the same. Hence, one can approximate the computation on those non-critical nodes or approximate the input data to those nodes. However, whether a node is critical to the output might be input dependent and varies from one execution to another, making this a challenging problem.

Any form of approximate computing will result in degradation in the quality of the computation result. That is why we have stated earlier that approximate computing techniques can only be applied to systems that can tolerate errors. Suppose the error caused by approximation goes beyond the level that the system can tolerate. In that case, such an approximation method cannot be used because it fails to deliver the required quality of service. Therefore, when considering approximate computing for energy-efficient design and implementation, it will be vital to guarantee that a user-specified quality of execution (or the level of errors during execution) is maintained. For example, when running a machine learning model for a classification task, the model or the execution could be approximated as long as the correct class label is assigned.

Designing energy-efficient systems with an approximation is a complicated process. The three questions mentioned above are among the main challenges. Failing to address them adequately would most likely end up in a failed design. As one can see, solutions to these questions are highly related to the applications that the system will perform. Hence there does not exist a one-fit-all solution. In the following, we highlight the key ideas of three sample approximation techniques for DFGs. The technical details will be elaborated on in the subsequent sections.

3.3 Approximation Techniques

In this subsection, we introduce three approximation techniques for DFGs, emphasizing their main ideas and how they answer the three previously discussed questions.

The first technique is proposed for systems such as multimedia data processing [10, 11]. It is based on the assumption that the same DFG will be executed periodically

(e.g., the frame decoding task when playing a video), and successful completion for every iteration is not required. In other words, it is acceptable for the DFG execution to produce inaccurate (or even incorrect) outputs for some iterations. Thus, the authors proposed taking advantage of this error tolerance and the uncertainties in the execution time by terminating certain iterations early to save energy. These iterations are selected based on the real-time execution time information to guarantee that the overall completion ratio will meet the user's statistical performance requirement. In terms of the three questions, such probabilistic design (1) approximates the entire DFG on selective iterations; (2) uses early termination of the execution once a long execution time is predicted; (3) provides probabilistic guarantees on the number of completed iterations to ensure performance.

The second technique considers a general DFG and is built on the observation that different inputs to the same node in a DFG may have significantly different impacts on the accuracy of the node's output [4]. That is to say, if one can identify this impact for each piece of data in a low-cost manner, the value of the "unimportant" data does not need to be as accurate as those "important" data; thus, the operations for generating these "unimportant" data can be approximated. The authors proposed a couple of low-cost runtime methods based on converting data to the logarithmic domain. The goal of the computation in the log domain is to distinguish, during the execution of a DFG, the critical subgraphs that produce "important" data and the non-critical subgraphs. In this estimate-and-recompute method, (1) the non-critical subgraphs in the DFG become approximable; (2) converting the expensive multiplication operation to the low-cost addition operation in the logarithmic domain is the proposed approximation method; (3) threshold values are set to provide the required computation accuracy.

The last technique is designed for logarithms and other complicated operations (square root is another example). These operations are widely used and are in general implemented through iterations of the basic operations such as addition and multiplication [28]. Comparing to the basic operation nodes in DFG, such complicated operations consume much more energy, but their impact on the output accuracy may be the same as basic operations. For example, in the simple operation $a+b$, a and b will impact the output's accuracy even if a is the output of an adder and b comes from a logarithm operation. Determining the energy and accuracy trade-off of these operations will provide a guideline for approximation. We perform a thorough analysis of these complex operators' error and energy models and deduce their impact on the DFG outputs. More specifically, in this approach, (1) the computation inside the implementation of complex operators is approximable; (2) the number of loops or iterations in the complex operators can be reduced to save power and energy; (3) the thorough error analysis guarantees that results from the approximated complex operators give the required accuracy while achieving a maximal reduction of power and energy.

4 Probabilistic Design for Multimedia Systems

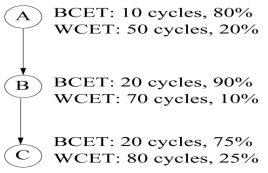
For the current multimedia systems, most present techniques are based on worst- or average-case scenarios [3, 18, 8]. The worst-case performance requirement usually leads to overdesigning because each single data point should be considered and taken good care of to ensure the successful completion under all the cases. However, the average-case performance requirement usually overly relaxes the hardware constraint and cannot guarantee the completion of half the cases. Given the disadvantages of these two requirement settings, designers start to consider providing a statistical completion ratio guarantee for a multimedia system [26, 9, 13]. Based on this new requirement, a probabilistic design is proposed to relax the hardware constraints and avoid the overdesigning [10, 11]. The technique utilizes the minimum effort to quickly check whether the current task can be completed on time and decide whether the remaining steps are worth running early.

This section is organized as follows. We will use a toy example to explain why probabilistic designs are needed or what benefits can be achieved via such designs. Next, we will talk about the architecture for the probabilistic design and discuss the whole workflow and every part of it. Finally, we will show the design details for the two main processes inside the workflow, probabilistic timing performance profiling, and estimation, and offline and online resource management.

4.1 An Illustrative Example

Consider an application that should be executed repetitively under different data points but could tolerate a certain degree of execution failures. A probabilistic design is proposed for such an application. For example, a target application that needs to be approximated consists of three sequential nodes, A, B, and C, as shown in Figure 2. It should be noted that Figure 2 is not a complete representation of a DFG because no specific operations are provided. Still, it is sufficient to calculate the overall execution time since the computation sequence is provided. Suppose each task has two possible execution times, one for the best-case execution time (BCET) and the other for the worst-case execution time (WCET). It is easy to know that there are eight possible execution times for the execution of all three tasks when the sum for any combination differs from each other. All the eight scenarios are listed in Figure 2. Besides, the real execution time (RET) and the probability of occurring this scenario are provided for each scenario.

Suppose that we set the execution time deadline to be 200 cycles and repetitively execute the application described above. In that case, all the iterations will be finished on time since the longest execution time for this design is 200 cycles. However, if only 70% of iterations are required to be executed on time, there is no need to set the deadline 200 cycles, and 100 cycles suffice. That means termination can be decided at 100 cycles when the execution fails to finish. Besides, termination decision can be made even earlier. For example, if the total execution time for node A and node B is



	A	B	C	RET	Prob
1	10	20	20	50	54.0%
2	50	20	20	90	13.5%
3	10	70	20	100	6.0%
4	10	20	80	110	18.0%
5	50	70	20	140	1.5%
6	50	20	80	150	4.5%
7	10	70	80	160	2.0%
8	50	70	80	200	0.5%

Fig. 2 An application with three sequential tasks and all the execution time scenarios (source of the figure: [11]).

more than 80 cycles, it is not possible to finish all the three tasks on time. Therefore, if A and B cannot finish in 80 cycles, we could stop the current iteration early.

Based on the analysis of the toy example, we could obtain two basic conclusions. The first conclusion is that the execution time can be shortened if we do not need 100% completion ratio for the current application. The second conclusion is that the application can be stopped at early stages to avoid unnecessarily execution of the following nodes when time is insufficient.

4.2 Probabilistic Design Overview

The probabilistic design methodology is illustrated in Figure 3. Similar to the toy example, designers should start with the application, performance requirements, and a list of target system architectures. The application can be represented by a DFG, which consists of a set of nodes and the edges between them where nodes and edges represent operations and data dependencies, respectively. For the performance requirements, two main metrics should be paid attention to, one is the statistical completion ratio, and the other is timing constraints. With the profiling tools, we discover all the scenarios with different execution times and calculate the probabilities of their occurrence. After estimating the probabilistic timing performance, we can determine whether the current setting can meet the completion ratio constraint or not. If the answer is no, we need to change either the software optimization or the hardware configuration and repeat the profiling analysis with the new setting. However, if the answer is yes, we could step forward to the resource management process. Two specific techniques are utilized in this process. The first technique is static, allocating minimum resources to each node offline, and the second technique is dynamic, allocating resources at runtime by developing real-time schedulers. Finally, system synthesis and evaluation will be done on the whole application to ensure all the performance requirements will be met under the current hardware, software, and scheduler settings. After the evaluation process, the application can be put into use. To summarize, probabilistic design mainly consists of probabilistic timing performance profiling and estimation and offline and online resource management, which will be discussed in detail in the following subsections.

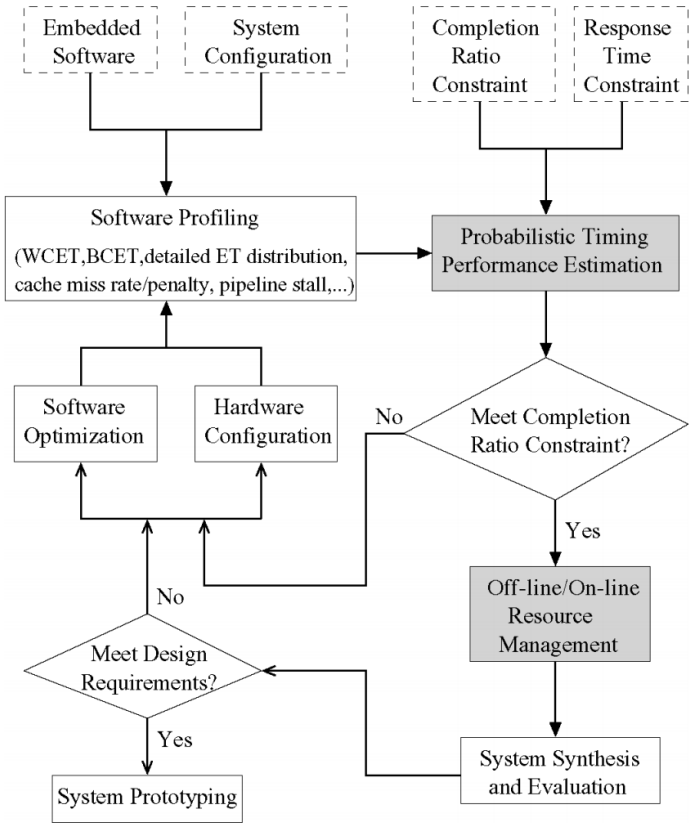


Fig. 3 Design flow in the probabilistic design methodology (source of the figure: [11]).

4.3 Probabilistic Timing Performance Profiling and Estimation

There are several papers on the probabilistic timing performance estimation for soft real-time system designs [26, 9, 13]. The general assumption is that each task's execution time can be described by a discrete probability density function that can be obtained by applying path analysis and system utilization analysis techniques [19]. Following the previous works, we consider the following scenario.

Consider a data flow graph $G = (V, E)$ given an application, where V denotes the set of nodes and E denotes the set of edges. The execution time for each node is described by a discrete probability density function as explained in the toy example. To be more specific, for each node v_i, it can be executed for k_i different times $t_{i,1}, t_{i,2}, ..., t_{i,k_i}$ under corresponding probabilities $p_{i,1}, p_{i,2}, ..., p_{i,k_i} | \sum_{l=1}^{k_i} p_{i,l} = 1$. That is to say, the probability of node v_i taking $t_{i,j}$ time to be completed is $p_{i,j}$.

The total completion time for the DFG G under the execution order $< v_1 v_2 ... v_n >$ can be calculated by summing up the individual execution time for each node e_i. If

we denote the total completion time by $C(< v_1 v_2 ... v_n >)$, it should equal to $\sum_{i=1}^{n} e_i$. M denotes the deadline constraint, or the maximum time provided to complete the application. G will be executed repetitively under different data points. If the execution time is longer than the deadline constraint $C(< v_1 v_2 ... v_n >) > M$, we consider the current iteration an failure. Otherwise, the iteration has been completed. Given $N >> 1$ iterations, suppose that K can denote the number of completed iterations, the completion ratio for the given application with deadline constraint M is $Q = K/N$. Other than the timing performance requirement, another performance requirement, the completion ratio constraint, can be denoted by Q_0. We say the completion ratio constraint satisfies over N iterations under M timing constraint if $Q \geq Q_0$.

Theorem 1 *The maximum achievable completion ratio is given by:*

$$Q^{max} = \sum_{\sum_{i=1}^{n} t'_{i,j_i} \leq M} \prod_{i=1}^{n} p_{i,j_i} \tag{2}$$

where the sum is taken over the execution time combinations that meet the deadline constraint M and the product computes the probability that each combination occurs.

Based on the theorem above, $Q^{max} < Q_0$ implies the fact the completion ratio constraint requirement can never be achieved under the current hardware and software settings. Thus designers should consider modifying the system settings and calculate the maximum achievable completion ratio again.

However, estimating the maximum completion ratio as shown in Equation 2 is computationally expensive since there are multiple nodes in a DFG and there are multiple execution times for a specific node. Even if there are only two possible execution times, BCET and WCET, the number of execution time combinations is 2^n, which increases exponentially with the number of nodes. Therefore, the following polynomial heuristic has been proposed to achieve a fast estimate of the completion ratio.

The execution times for a specific node can be sorted as $t_{i,1} < t_{i,2} < ... < t_{i,k_i}$ and the prefix sum or the occurrence probability can be calculated by:

$$P_{i,l_i} = \sum_{j=1}^{l_i} p_{i,j} \tag{3}$$

which represents the probability when the execution time at node v_i does not exceed t_{i,l_i}. For each node v_i, a specific time such as t_{i,l_i} can be allocated. Based on this setting, the completion ratio can be given by:

$$Q = \prod_{i=1}^{n} P_{i,l_i} = \prod_{i=1}^{n} \sum_{j=1}^{l_i} p_{i,j} \tag{4}$$

A greedy algorithm is utilized for fast estimate n the completion ratio. First, the allocated execution time for each node could be its WCET, which makes sure $Q = 1$ and any completion ratio constraint can be achieved since $Q_0 \leq Q = 1$. If the allocated time for node v_i is reduced from t_{i,l_i} to $t_{i,(l_i-1)}$, based on Equation 4, the completion ratio will be reduced by $\frac{P_{i,(l_i-1)}}{P_{i,l_i}}$. Besides, the total allocated time will be reduced by $t_{i,l_i} - t_{i,(l_i-1)}$. Therefore, we iteratively cut the time slot for node v_j which yields the largest $(t_{j,l_j} - t_{j,(l_j-1)})\frac{P_{j,(l_j-1)}}{P_{j,l_j}}$ as long as the completion ratio is greater than Q_0. By cutting the time via this greedy-selection algorithm, the required completion ratio constraint can be achieved. If the total allocated time can reach the deadline constraint M, we say that the polynomial heuristic can conclude that the required Q_0 is achievable under the current hardware and software settings.

4.4 Offline and Online Resource Management

After determining that the current hardware and software setting can lead to a satisfying approximate design for the application under M deadline constraint and the Q_0 completion ratio constraint, the next step is to carefully consider managing the resources to save the most cost. We will first discuss a naive approach and then propose another improved scheduling algorithm.

Let us first consider *a naive best-effort approach*. It is very straightforward to execute all the nodes at the highest voltage and the highest speed until the deadline M is reached. This approach can reach the maximum achievable completion ratio, but it does not help save more energy costs; thus, we call it the naive best-effort approach. For this approach, we need to compute the energy consumption for comparison with other approaches. Among N iterations of execution for the whole DFG, K of them can be completed before the deadline. Suppose the completion time can be denoted by $C_i (1 \leq i \leq K)$. If the power dissipation at the reference voltage is P_{ref}, the energy consumption over N iterations can be calculated by:

$$E = P_{\text{ref}}(\sum_{i=1}^{K} C_i + (N - K)M) \tag{5}$$

Another approach, *online best-effort energy minimization algorithm*, is proposed for systems whose supply voltage can be switched at runtime. The system can switch the voltage at runtime to save more energy consumption while satisfying the same completion ratio. The energy can be saved on two occasions. On the one hand, if the completion occurs earlier than expected, we could allocate more time for each node by running the application at the lower voltage. On the other hand, if the application cannot be finished as expected, we could terminate the application earlier to avoid much power waste.

Before the algorithm details are introduced, let us start with the notations. For each node v_i in the execution sequence $< v_1 v_2 ... v_n >$, we define its latest completion

time T_{l_i} and earliest completion time T_{e_i}. For the last node v_n in the application, both the latest and earliest completion times are set to be M. For other nodes before v_n, their T_{l_i} and T_{e_i} depend on the shortest execution time and the longest execution time for the following nodes, respectively, which can be represented by the equations below.

$$T_{l_n} = T_{e_n} = M \tag{6}$$

$$T_{l_i} = T_{l_{i+1}} - t_{i+1,1} \tag{7}$$

$$T_{e_i} = T_{e_{i+1}} - t_{i+1,k_{i+1}} \tag{8}$$

We will discuss the online best-effort energy minimization algorithm with the notations mentioned above. The current node under execution is v_i and its actual execution time at the reference voltage in jth iteration is $e_{i,j,ref}$. If the current node cannot be finished on time, which can be represented by $t + e_{i,j,ref} > T_{l_i}$, the current iteration can be terminated early to save the energy. If the current node can be finished earlier than expected, which can be represented by $t + e_{i,j,ref} < T_{e_i}$, the voltage can be scaled so that the execution of the current node can be finished at T_{e_i}, which still leaves sufficient time for the following nodes.

5 Estimate and Recompute During DFG Execution

One method to fulfill the approximate computing for data flow graphs is to identify the non-critical computations by analyzing their impacts on the output accuracy. If the outputs are not sensitive to some nodes, these nodes do not need to be calculated accurately. Based on this idea, several previous works [21, 23, 22] try to target the non-critical parts by using the training data or by data range tuning and interval arithmetic. However, all these methods are working offline regardless of every input, as we call "static." Therefore, a runtime approximation paradigm is proposed in [4]. The proposed framework quickly estimates the impact of each input to output accuracy by converting the floating-point numbers to logarithmic numbers. With the fast estimate results, two algorithms are proposed to decide whether specific nodes need to be computed accurately or approximately.

5.1 Conversion from Floating-Point to Logarithmic

Two different types of floating point formats are widely used in multiple applications, single precision floating point and double precision floating point. Double precision is more likely to be applied to systems with high requirements on the precision, which are not suitable for approximate computing. Therefore, we mainly focus on the single

precision floating point format. Based on the IEEE 754 standard, single precision numbers are composed of 32 bits, a sign bit, 8 exponent bits, and 23 mantissa bits. The value can be computed by

$$\text{num} = \text{sign} * (1 + \text{mantissa}) * 2^{(\text{exponent}-127)} \tag{9}$$

When ignoring the sign bit, Equation 9 can be rewritten as $x = 1.m * 2^e$ where the dot represents the radix point. Let x_l be the logarithmic representation value of x and the value for x_l can be compuated by

$$x_l = \log_2(x) = \log_2(1.m * 2^e) = \log_2(1.m) + e \approx m + e = e.m \tag{10}$$

Based on Equation 10, we can obtain the logarithmic value x_l by directly exploiting the bits in single-precision floating-point data format and truncating the least significant bits in mantissa. At the same time, if we would like to recover the floating-point format from the logarithmic representation, we could pad "0"s to the end. The conversion process between logarithmic representation and the floating-point data format has been illustrated in Figure 4.

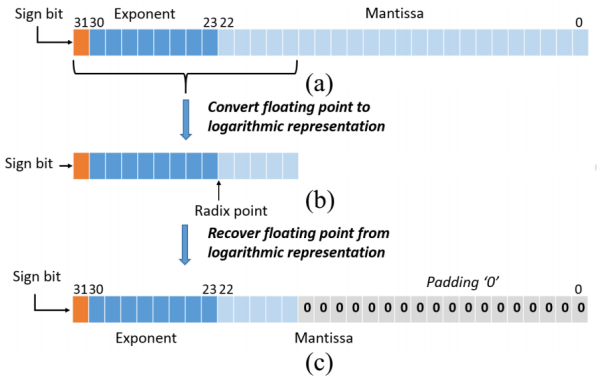


Fig. 4 Conversion of floating point formation between linear domain and log domain (source of the figure: [4]).

5.2 Arithmetic Operations in Logarithmic Representations

When numbers are represented in the logarithmic domain, the arithmetic operations should change accordingly. Therefore, we will discuss how the converted data can contribute the arithmetic operations. Intuitively, multiplication and division become easier to compute in the log domain, while addition and subtraction become more

complicated. Based on this intuition, the operations can be divided into accurate and approximate conversion.

Accurate conversion refers to the operations which can be processed directly without providing any error compensation. The common operations of this type include multiplication, division, square root, and power.

- For multiplication where $S = A * B$, the logarithmic value of the product can be calculated by $S_l = \log_2 S = \log_2(A * B) = \log_2 A + \log_2 B = A_l + B + l$. That is, the product in log domain can be obtained by adding the logarithmic operands.
- For division where $S = A/B$, it is easy to get $S_l = A_l - B_l$. That is, the quotient in log domain can be obtained by subtraction of the logarithmic operands.
- For square root where $S = \sqrt{A}$, $S_l = \log_2 S = \log_2 \sqrt{A} = \frac{1}{2}\log_2 A = \frac{1}{2}A_l = A_l >> 1$. That is, square root can be transformed into shifting.
- For power where $S = A^n$, $S_l = \log_2 A^n = n \log_2 A = nA_l$. That is, power can be transformed into one multiplication with integers.

The common characteristic for the operations mentioned above is that their counterpart computation in log domain is much cheaper than their original version in floating point domain.

Approximate conversion refers to the much more complicated operations in the log domain, such as addition and subtraction. Since the goal of the log domain is to identify the critical subgraph and the non-critical subgraph via fast estimation, there is no need to generate accurate results as long as the goal task can be finished as expected. Therefore, we introduce a novel addition and subtraction estimation with an error compensation technique. Consider addition as an example and suppose both the operands A and B are positive ($A > B$). The sum can be calculated by

$$\begin{aligned} S = A + B &= 2^{A_l} + 2^{B_l} = 2^{A_l}(1 + 2^{B_l - A_l}) \\ &\approx 2^{A_l}(1 + 2^{\text{round}(B_l - A_l)} = 2^{A_l}(1 + 2^{-ed}) \end{aligned} \tag{11}$$

where ed denotes the exponential difference. Therefore, the logarithmic value can be calculated by

$$\begin{aligned} S_l &= \log_2 S = \log_2 2^{A_l}(1 + 2^{-ed}) = \log_2 2^{A_l} + \log_2(1 + 2^{-ed}) \\ &\approx A_l + 2^{-ed} \end{aligned} \tag{12}$$

It should be noted that A_l has only 5 bits in the fraction part. Therefore, if $ed > 5$, $S_l = A_l$, and if $ed \leq 5$, $S_l = A_l + 1 >> ed$. Based on this technique, the results of addition and subtraction under log domain can be estimated quickly.

5.3 Noncriticality Truncation

Error Resilient and Sensitive Operations. For a DFG, we classify the operations into two types: (1) the error-sensitive operations and (2) the error-resilient operations.

Since each node in the DFG represents one operation, the nodes with error-sensitive operations are defined as error-sensitive nodes, denoted by n^s. Error-sensitive operations include multiplication, division, and exponentiation because small changes on the operands can cause a significant difference in the output. Besides, the nodes with error-resilient operations are defined as error-resilient nodes, denoted by n^r. Error resilient operations include addition, subtraction, and comparison because the dominant operand with a larger absolute value has a greater impact on the output. The dominant input and the minor input are denoted by I_d and I_m, respectively. More difference between the two operands means more resiliency for the operand with a smaller absolute value. This error-resilient feature is utilized to identify the non-critical inputs as well as their branches.

Noncriticality Definition and Classification. The non-critical input only occurs for error-resilient nodes. Since all the inputs greatly impact the output precision, they are all critical inputs for error-sensitive nodes. For error-resilient nodes, an input I_m is a noncritical input in log domain if and only if $f(I_d - I_m) \geq \delta$, where

$$f(x) = \begin{cases} x & \text{for } \mathrm{add}_l/\mathrm{sub}_l \\ \mathrm{abs}(x) & \text{for comparative operations} \end{cases}$$

δ represents the threshold designers use to control the computational quality. For example, $\delta = 1$ means that the dominant input is twice as large as the minor input. The larger δ means more difference between the two operands and less criticality for the I_m. After the threshold for the DFG is determined, the corresponding non-critical operands can be identified as well.

Truncation and Recomputation. After identifying the non-critical input given an error-resilient node n_i^r in the log domain, the steps to produce the approximate results are listed below. The key idea is to replace the non-critical operand with the estimated value in the log domain and recompute critical operands accurately in the linear domain.

- Cut off n_i^r's noncritical parent branch.
- Replace the noncritical operand with the estimated value I_m in log domain.
- Convert I_m in log domain to the value in linear domain.
- Recompute n_i^r's critical parent branch in linear domain accurately.
- Compute node n_i^r accurately.

Figure 5 presents a motivation example of this truncation and recomputation step.

Error Analysis. Next, we deduce a theoretical error analysis of the truncation and recomputation steps. Consider addition as an example and denote the accurate input values for n_i^r by v_m and v_d. Since l_m and l_d are their logarithmic representations, we can get $v_m = 2^{I_m}$ and $v_d = 2^{I_d}$. If the estimated error for I_m and I_d are ϵ_m and ϵ_d. The recovered value for the minor input is $2^{I_m+\epsilon_m}$. Therefore, the error rate of node n_i^r is given by

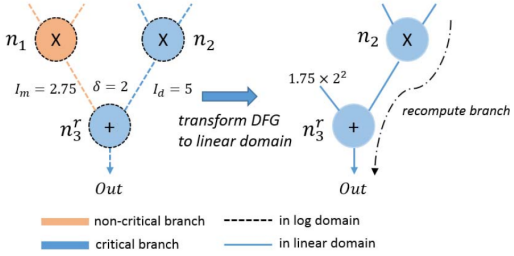


Fig. 5 Example of truncation and recomputation (source of the figure: [4]).

$$
\begin{aligned}
er &= \frac{O_{\text{apx}} - O_{\text{acc}}}{O_{\text{acc}}} = \frac{(2^{I_m+\epsilon_m} + v_d) - (v_m + v_d)}{v_m + v_d} \\
&= \frac{2^{I_m+\epsilon_m} - v_m}{v_m + v_d} = \frac{2^{I_m+\epsilon_m} - 2^{I_m}}{2^{I_m} + 2^{I_d}} \\
&\leq \frac{2^{\epsilon_m} - 1}{2^{\delta-\epsilon_d+\epsilon_m} + 1}
\end{aligned}
\tag{13}
$$

where O_{apx} and O_{acc} are the approximate and accurate outputs. Assume $\epsilon_m - \epsilon_d \approx 0$, then

$$
er \leq \frac{2^{\epsilon_m} - 1}{2^{\delta} + 1} \tag{14}
$$

With a proper threshold value, the error rate can be controlled within a small range.

5.4 Runtime DFG Approximation Algorithms

Although we could determine the criticality of the parent branches of a given node n_i^r, we cannot remove all the non-critical nodes directly because the node n_k might be in the non-critical branch of n_i^r and the critical branch of n_j^r at the same time. Simply removing the node n_k will not cause a significant difference in the output of n_i^r but is likely to cause unacceptable errors in the output of n_j^r. Therefore, considering the scenario described above, we proposed two graph truncation algorithms.

The *GlobalCut* algorithm transforms the problem of removing the non-critical node to minimally reserving the critical nodes on the path from the primary inputs to the primary outputs. The algorithm starts from the output nodes and finds all the critical nodes in the path back to the input nodes. To traverse the graph, we could initialize a queue with ith output node O_i. For the first node in the queue, do the following steps recursively until the queue is empty: (1) figure out all the critical parent nodes and (2) mark them as executable nodes. After finishing the algorithm, the minimally reserved critical nodes are discovered. Approximate output can be calculated by simply executing these critical nodes in the linear domain.

Unlike the GlobalCut, which directly ignores the non-critical nodes, *LocalCut* algorithm considers them because the errors can go up when being propagated to the outputs of the graph. Therefore, LocalCut Algorithm starts from the input nodes and goes forward. First of all, estimate the graph in the log domain until meeting the next error-resilient node n_i^r. Then, do the truncation and recomputing steps described in the previous section. The output for node n_i^r can be obtained from its recomputed results. The key idea behind this is to recompute the critical nodes accurately in the linear domain while replacing the remaining non-critical nodes with their estimated values in the log domain.

6 Approximate Logarithms by Bit-Width Optimization

Various truncation methods have been proposed for basic operations such as addition [20] and multiplication [27], but its application to logarithm is not well explored. The logarithm is more highly energy-hungry than these basic operations because it requires many iterations of the basic operations to achieve high accuracy. For example, a common way to implement logarithm is to use iterations, which we will elaborate on later. Besides, the logarithm is a fundamental operation in statistical models and machine learning tasks such as variational inference, Bayesian methods, and neural networks on many resource-constrained systems. Therefore, implementing energy-efficient logarithm calculation is highly desirable. The recent explosion of data size, computation cost, and the approximative nature of these tasks provide an ideal environment for approximate computing. We will explore the feasibility of using approximate computing for energy-efficient logarithm operation and demonstrate the energy savings it achieves with little accuracy loss [28].

The critical challenge for approximate computing in trading off performance degradation for energy saving is estimating errors in the logarithmic operation and how it propagates and amplifies in the final results of the entire program or application. Truncation, an approximation approach whose introduced error can be expressed as a function of the number of approximate bits, provides a sound basis for error analysis and hence the quality control of the results. For instance, the bit-width optimization problem has been formulated to analyze the error behavior of truncation and to determine the optimal number of approximate bits for each operand in a program without causing significant accuracy degradation to the result. This problem has been well-studied for basic operations such as addition and multiplication. Therefore, we will investigate how to achieve energy efficiency and quality control of a program with logarithm operations through bit-width optimization.

6.1 Error Analysis for Logarithms

We consider IEEE 754, the standard format of floating-point arithmetic, as the way of representing data. As shown in Figure 6, the 32-bit IEEE 754 format for a floating-point data a consists of three parts: a sign bit s_a, an 8-bit exponent e_a and a 23-bit mantissa m_a.

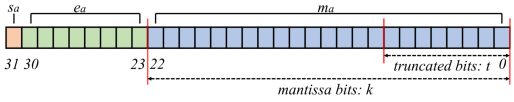


Fig. 6 IEEE 754 format with the last t bits to be truncated during the approximate computation (source of the figure: [28]).

The error introduced to the data comprises three parts, truncation error, propagation error, and calculation error. Truncation error is due to the direct modification of data, while propagation error is due to the propagation of truncation error through each operation in the program. Calculation errors are those from the imperfection of hardware or software implementation and are considered rare.

Before studying how it affects calculation results for an operation like logarithm, we first analyze how truncation affects operand's value or truncation error. Assume that the least significant t bits are truncated from the total k bits in the mantissa; the induced error range can be given by

$$x_a \in [-(\frac{1}{2^{k-t}} - \frac{1}{2^k}) * 2^{e_a}, (\frac{1}{2^{k-t}} - \frac{1}{2^k}) * 2^{e_a}] \tag{15}$$

Since k is a fixed number such as 32 or 64, the range of truncation error x_a only depends on the number of bits to be truncated. However, the exact value of truncation error varies as the input value varies. In order to compute error variance, we assume that the value of last t bits in operand a is uniformly distributed and denote the distribution probability as p_a. Thus, the variance of truncation error can be computed by

$$\sigma_a^2 = \sum x_a^2 p_a = 2^{2(e_a-k)} * \frac{(2^t-1)2^t}{3} \tag{16}$$

The next step is to analyze how this error will be transmitted to calculation results or propagation errors. Since some prior works [25, 24, 6] have already deduced propagation error caused by addition, multiplication, and shift operations, what we focus on in this paper is the logarithm operation. Suppose the output for logarithm is denoted by out, and its variance is represented by σ_{out}^2. The logarithm output variance, as a function of operand variance σ_a^2, is given by $\sigma_{\text{out}}^2 = (\partial out/\partial a)^2\sigma_a^2$. After substituting σ_a^2 with 16, the final truncation and propagation error variance for logarithm is shown below.

$$\sigma_{\text{out}}^2 = \frac{1}{a^2} * \sigma_a^2 = \frac{1}{(m_a * 2^k)^2} * \frac{(2^t - 1)2^t}{3} \tag{17}$$

which suggests that the maximal error variance is achieved when mantissa $m_a = 1$ for any given value of t.

6.2 Energy Analysis for Logarithms

In this subsection, we discuss the impact of data truncation on energy savings for the logarithm operation. As shown in Figure 7, the logarithm is implemented with iterations of basic arithmetic units. Therefore, energy savings can be achieved by low-precision computation units (e.g., adder or multiplier) and fewer iterations required for a specific error budget.

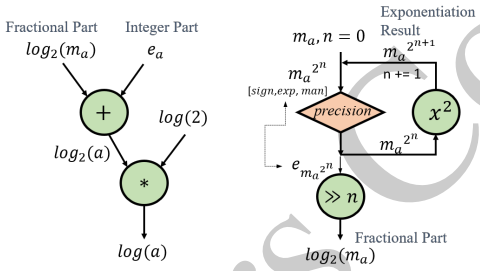


Fig. 7 Logarithm calculation flow chart (source of the figure: [28]).

First, we illustrate the implementation details for the logarithm. [1] proposes an approach to calculate logarithm by exploiting floating-point data exponent. As shown in Figure 7 (a), the problem of computing $\log a$ (base e) can be easily transformed into computing $\log_2 a$. On the one hand, the integer part e_a can be directly extracted from IEEE 754 format. On the other hand, the fractional part $\log_2 m_a$ can be computed as illustrated in Figure 7 (b). The core algorithm is to apply repeated square operations to m_a for exponentiation result $m_a^{2^n}$, where n denotes the number of loops executed. Since $m_a^{2^n}$ is also a floating-point number, its logarithm base 2 can be approximated by the exponent value $e_{m_a^{2^n}}$. Thus, the fractional part is given below, and it is precise up to $1/2^n$ because the first n bits are accurate.

$$\log_2 m_a = \frac{1}{2^n} * \log_2 {m_a}^{2^n} \approx \frac{1}{2^n} * e_{m_a^{2^n}} = e_{m_a^{2^n}} >> n \tag{18}$$

Next, we analyze how the number of bits to be truncated, t, affects the number of loops executed, n, for the logarithm operation given an error budget. Since the first n bits in the fractional part are accurate, calculation error x_{calc} ranges from 0 to $1/2^n$. Given the uniform error distribution, calculation error variance is given by

$$\sigma^2_{\text{calc}} = \int x^2_{\text{calc}} p_{\text{calc}} - \mu^2 = \frac{1}{12} * (\frac{1}{2^n})^2 \tag{19}$$

To stick to our prior assumption that calculation error caused by hardware or software is insignificant compared with truncation and propagation error, we choose n such that calculation error variance is restricted, as shown in 19 where δ is the hyperparameter which the user can define.

$$\sigma^2_{\text{calc}} \leq \delta \sigma^2_{\text{tNp}} \tag{20}$$

After substituting σ^2_{calc} and σ^2_{tNp} with Equation 19 and 17, respectively, we obtain Equation 21. It shows that at least $k - 1/2 * \log_2 4\delta$ loops are required to fully exploit all the significant digits when no bit is truncated. When t LSB bits are truncated, t loops can be omitted without introducing noticeable calculation error.

$$n \geq k - t - \frac{1}{2} \log_2 4\delta \tag{21}$$

Finally, we consider the effect of truncation on the overall energy consumption of the logarithm. The minimal number of iterations required for computing an acceptable logarithm result is determined in Equation 21. Given the total number of loops n, the energy consumption for logarithm operation is linearly dependent on n. Besides, as shown in Figure 7 (b), each iteration includes several arithmetic operations. Due to the reduced data length, some operations can be simplified, which also creates energy savings. After combing the above two effects, the total energy consumption for the truncated logarithm operation is given in 22, where E_{m_t}, E_{s_t}, E_{add} denote energy for truncated floating-point multiplier, shifter and integer adder, respectively.

$$E_{\text{log\_t}} \geq (k - t - \frac{1}{2} \log_2 4\delta) * (E_{\text{m\_t}} + E_{\text{s\_t}} + E_{\text{add}}) \tag{22}$$

6.3 BWOLF System Structure

We propose the BWOLF (Bit-Width Optimization for Logarithmic Function) system for programs with logarithm and other basic arithmetic operations. BWOLF considers a program in the popular data flow graph model and utilizes sequential quadratic programming to determine the optimal number of approximate bits for each operand to minimize energy consumption.

The proposed BWOLF system is illustrated in Figure 8. The system requires three inputs: a target program, the input range, and an upper bound of error. The given program should have a fixed number of inputs and outputs, exclude conditionals and be deterministic where output values will not change given the same input values. The data range for input is needed for the estimation of maximal output error variance. For example, the maximal error variance for a truncated input depends exponentially

on the maximal value of exponent (see Equation 16). The error budget is an upper bound for the output error deviation σ (the square root of the error variance σ^2).

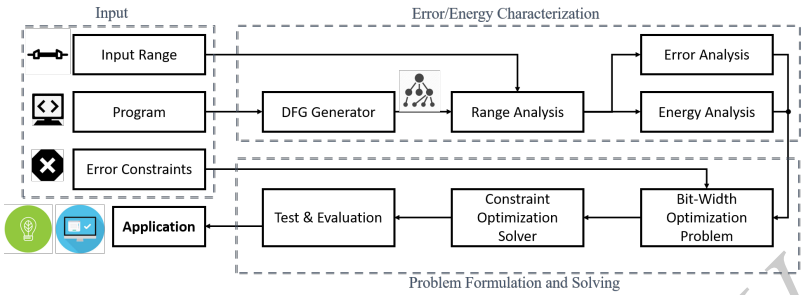


Fig. 8 Overview of BWOLF system structure (source of the figure: [28]).

The *characterization stage* figures out error and energy models for a program. Suppose the number of approximate bits for each data is given. In that case, an error model can determine the maximal output error variance when inputs change, while an energy model can determine the energy required to produce results. There are three steps to obtain them. First, based on the program, the data flow graph (DFG) generator produces a directed graph whose nodes represent operators and edges represent data. The next step is to conduct a range analysis on the DFG to obtain the minimum and maximum value for each data, which provides the basis for further investigation on maximal error variance for outputs. Finally, based on the topological order in the DFG and the range for each data, error and energy analyses will determine the relationship between the number of approximate bits for each data and the maximal output error variance and total energy consumption.

To obtain error and energy models for the whole program, we utilize approximate adder [20]‘ and multiplier [27] whose error and energy behavior has been thoroughly analyzed. The error and energy behavior for the logarithm node is illustrated in the previous sections.

Let us consider a data flow graph that is composed of addition, subtraction, multiplication, and logarithm operations as shown in Figure 9. Each node in the graph represents an operation, and each edge represents a value passed between operations. For a directed graph G, there are D edges and V nodes which are denoted as $D_i (1 \leq i \leq D)$ and $V_j (1 \leq j \leq V)$. The edges are sorted so that the input edge of a node has a smaller index than its output edge. The number of bits to be truncated for each edge is represented as $T_i (1 \leq i \leq D)$.

Since each edge in DFG can be truncated, truncation error σ_t^2 will be introduced to the whole flow. This error will then be propagated from a node's input edge(s) to its output edge. Such propagation transforms the error into truncation and propagation error, which is denoted by σ_{tNp}^2 (tNp: truncation for input edges and propagation for nodes). For example, σ_a^2 and σ_{out}^2 given in Equation 16 and 17 are indeed σ_t^2 and σ_{tNp}^2 for logarithm node. It should be noted that the outputs for prior nodes are the

inputs for the following nodes, which are also eligible for truncation. As shown in the zoomed in box in Figure 9, we denote the error variance caused by truncation of a node's output edges as σ^2_{next} . More precisely, it is denoted as σ^2_{next} for logarithm but denoted as σ^2_t for subtraction operation.

To obtain error model or maximal error variance for graph's outputs (i.e. D_{13} in Figure 9), the first step is to calculate the truncation error for graph's inputs (i.e. D_1,D_2,D_7,D_8 in Figure 9). Next, for each node in the topological order in DFG, its maximal truncation and propagation error can be obtained based on node operation, operands' error variance, and range. The node's output data, as well as error variance, will then be passed to the next node. However, if the number of approximate bits for the node's output edge (e.g. T_3 for logarithm node V_1's output edge D_3) is quite small, σ^2_{next} will be much smaller than σ^2_{tNp}. In this case, the precision for current data (i.e. D_3) is still controlled by σ^2_{tNp} when extra energy is required for higher-precision subtraction operation. Therefore, we select the number of approximate bits (i.e. T_3) by ensuring σ^2_{next} is compatible with or even larger than σ^2_{tNp}. This inequation is added to Constraints(T). After going through all nodes in the graph, maximal truncation and propagation error variance for all outputs in the graph can be obtained. If there are multiple output nodes, we sum up the error variance of each node for final error variance Error(T). Besides, we calculate energy consumption for each node based on the number of approximate bits in its input edges. Summing up energy consumption for all nodes in the overall energy Energy(T) required for finishing computing the graph.

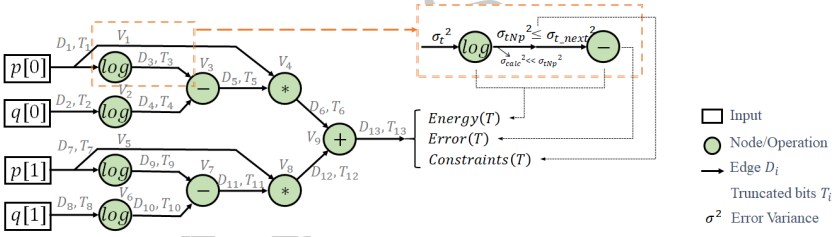


Fig. 9 DFG example of KL divergence for two Bernoulli distributions (source of the figure: [28]).

The *problem formulation and solving stage* aims at selecting the best approximate configuration which minimizes the energy consumption while keeping the maximal output error variance under constraint. The problem of bit-width optimization is seeking the best number of bits to be truncated for each data to minimize energy consumption when the error is within the user-defined range. As illustrated in the previous subsection, we have obtained error model Error(T), energy model Energy(T), and constraint functions Constraints(T). All of them depend on T, which represents the approximate configuration. Finding the best approximate configuration is formulated below, where ec represents the user's fixed error budget for σ.

$$\begin{aligned} \min_{T} \quad & \text{Energy}(T) \\ \text{s.t.} \quad & \text{Error}(T) \leq ec^2 \\ \text{and} \quad & \text{Constraints}(T) \end{aligned} \tag{23}$$

To solve the nonlinearly constrained optimization problem above, we adopt sequential quadratic programming [2] to find the optimal configuration. This algorithm utilizes a quadratic programming subproblem to approximate the original complex nonlinear problem and exploits the solution to the simplified problem to help generate a better approximate quadratic subproblem. The solutions for the sequence of subproblems finally converge to the optimal point for the original nonlinear problem.

7 Summary

With the increasing demand for resource-constrained end devices such as the Internet of Things, low-power system design remains an active and challenging research topic. Approximate computing is a promising technique to save power and energy for systems that are error-tolerant or error-resilient. This chapter provides the motivation of approximate computing on data flow graphs (DFGs) and discusses the main concerns in approximating DFGs. Since both the basic arithmetic units such as adders and multipliers and the input data of a DFG have been well-studied for approximation, we focus on several novel approaches that target the entire DFG or the underlying applications for approximation. The first technique, probabilistic design of multimedia systems, approximates the DFG execution by terminating the iterations with predicted long execution time early to save energy under the completion ratio constraint. The second technique, estimate-and-recompute, approximates floating-point multiplications by addition in the logarithm domain and then recomputes for the accurate value only for operations on the critical subgraphs. The last example demonstrates how the logarithm operation can be represented as a DFG and computed approximately by optimizing the bit-width of the input. Our goal in this chapter is to inspire novel approximate computing approaches that investigate the intrinsic nature of the specific application to achieve the optimal approximation strategy to trade computation accuracy for energy reduction.

References

1. Tms320 dsp development support reference guide. Tech. rep., Texas Instruments, 1998.
2. Boggs, P. T., and Tolle, J. W. Sequential quadratic programming. *Acta numerica 4* (1995), 1–51.
3. Eikerling, H.-J., Hardt, W., Gerlach, J., and Rosenstiel, W. A methodology for rapid analysis and optimization of embedded systems. In *Proceedings IEEE Symposium and Workshop on Engineering of Computer-Based Systems* (1996), pp. 252–259.

4. Gao, M., and Qu, G. Estimate and recompute: A novel paradigm for approximate computing on data flow graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 2 (2020), 335–345.
5. Gao, M., Wang, Q., Nagendra, A. S. K., and Qu, G. A novel data format for approximate arithmetic computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)* (2017), pp. 390–395.
6. Gao, M., Wang, Q., and Qu, G. Energy and error reduction using variable bit-width optimization on dynamic fixed point format. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2019), pp. 152–157.
7. Hashemi, S., Bahar, R. I., and Reda, S. A low-power dynamic divider for approximate applications. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2016), pp. 1–6.
8. Henkel, J., and Ernst, R. High-level estimation techniques for usage in hardware/software co-design. In *Proceedings of 1998 Asia and South Pacific Design Automation Conference* (1998), pp. 353–360.
9. Hu, X., Zhou, T., and Sha, E.-M. Estimating probabilistic timing performance for real-time embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 6 (2001), 833–844.
10. Hua, S., Qu, G., and Bhattacharyya, S. S. Energy reduction techniques for multimedia applications with tolerance to deadline misses. In *Proceedings of the 40th annual Design Automation Conference (DAC'03)* (2003), pp. 131–136.
11. Hua, S., Qu, G., and Bhattacharyya, S. S. Probabilistic design of multimedia embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 6*, 3 (2007), 15–es.
12. Jiang, H., Han, J., and Lombardi, F. A comparative review and evaluation of approximate adders. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI* (New York, NY, USA, 2015), GLSVLSI '15, Association for Computing Machinery, p. 343–348.
13. Kalavade, A., and Moghé, P. A tool for performance estimation of networked embedded end-systems. In *Proceedings of the 35th Annual Design Automation Conference* (New York, NY, USA, 1998), DAC '98, Association for Computing Machinery, p. 257–262.
14. Kianzad, V., Bhattacharyya, S. S., and Qu, G. Casper: an integrated energy-driven approach for task graph scheduling on distributed embedded systems. In *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)* (2005), pp. 191–197.
15. Liu, C., Han, J., and Lombardi, F. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)* (2014), pp. 1–4.
16. Liu, W., Lombardi, F., and Shulte, M. A retrospective and prospective view of approximate computing [point of view. *Proceedings of the IEEE*.
17. Liu, W., Qian, L., Wang, C., Jiang, H., Han, J., and Lombardi, F. Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Transactions on Computers 66*, 8 (2017), 1435–1441.
18. Madsen, J., Grode, J., Knudsen, P. V., Petersen, M. E., and Haxthausen, A. Lycos: The lyngby co-synthesis system. *Design Automation for Embedded Systems 2*, 2 (1997), 195–235.
19. Malik, S., Martonosi, M., and Li, Y.-T. S. Static timing analysis of embedded software. In *Proceedings of the 34th Annual Design Automation Conference* (New York, NY, USA, 1997), DAC '97, Association for Computing Machinery, p. 147–152.
20. Nannarelli, A. Tunable floating-point adder. *IEEE Transactions on Computers 68*, 10 (2019), 1553–1560.
21. Nepal, K., Li, Y., Bahar, R. I., and Reda, S. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)* (2014), pp. 1–6.
22. Riehme, J., and Naumann, U. Significance analysis for numerical models. In *1st Workshop on Approximate Computing (WAPCO)* (2015), pp. 0278–0070.

23. Roy, P., Ray, R., Wang, C., and Wong, W. F. Asac: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems* (New York, NY, USA, 2014), LCTES '14, Association for Computing Machinery, p. 95–104.
24. Sengupta, D., Snigdha, F. S., Hu, J., and Sapatnekar, S. S. Saber: Selection of approximate bits for the design of error tolerant circuits. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2017), pp. 1–6.
25. Snigdha, F. S., Sengupta, D., Hu, J., and Sapatnekar, S. S. Optimal design of jpeg hardware under the approximate computing paradigm. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2016), pp. 1–6.
26. Tia, T.-S., Deng, Z., Shankar, M., Storch, M., Sun, J., Wu, L.-C., and Liu, J.-S. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings Real-Time Technology and Applications Symposium* (1995), pp. 164–173.
27. Wires, K. E., Schulte, M. J., and McCarley, D. Fpga resource reduction through truncated multiplication. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications* (2001), FPL '01, Springer-Verlag, p. 574–583.
28. Xu, Q., Sun, G., and Qu, G. Bwolf: Bit-width optimization for statistical divergence with -logarithmic functions. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2020), pp. 165–172.