On Pareto-frontier Approximate Computing for Many-core Systems

Xinyue Hou
School of Software Engineering
South China University of Technology
Guangzhou,China
se_xinyue.hou@mail.scut.edu.cn

Xiaohang Wang
School of Software Engineering
South China University of Technology
Guangzhou, China
xiaohangwang@scut.edu,cn

Maurizio Palesi
Department of Electrical, Electronics and Computer Engineering
University of Catania
Catania, Italy
maurizio.palesi@dieei.unict.it,

Amit Kumar Singh
School of Computer Science and Electronic Engineering
University of Essex
Colchester, United Kingdom
a.k.singh@essex.ac.uk

Yingtao Jiang
Department of Electrical and Computer Engineering
University of Nevada
Las Vegas, USA
jingtao.jiang@unlv.edu

Mei Yang
Department of Electrical and Computer Engineering
University of Nevada
Las Vegas, USA
mei.yang@unlv.edu

Letian Huang
School of Electronic Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
huanglt@uestc.edu.cn

***Abstract*—Approximate computing is an emerging paradigm that aggressively improves performance or reduces energy consumption by sacrificing computation quality for error forgiving applications. Various approximate techniques, including loop truncation, approximate communication, etc. have been proposed. Previous works focus on optimization using only one approximation knob. However, we have observed that simultaneously optimizing with multiple approximation knobs leads to a large search space and is more likely to find better solutions. Therefore, in this paper, we first develop application models for performance, error, and power, followed by formulation of an optimization problem to maximize system performance under error and power constraints, using three approximation knobs, which are loop truncation, data dropping, and computational precision scaling. In order to solve the problem efficiently, a lightweight algorithm based on interior point algorithm is proposed. Experimental results show that, compared to state-of-the-art approximate approaches, the proposed scheme can reduce the execution time by as much as 33.1%. The overhead of the proposed method is low, making it a suitable approximate scheme for future many-core systems.**

Keywords—approximate computing, many-core systems, coordinated control, networks-on-chip

I. INTRODUCTION

In error-forgiving applications like image/video processing, machine learning, and data mining etc., an approximate result is acceptable if it satisfies application quality constraint. For error-forgiving applications, the output quality can be traded-off for higher performance and/or lower energy consumption by tolerating errors. Compared to traditional power management schemes, approximate computing can further reduce execution time or energy consumption.

In the literature, there are many approximate techniques for computation, on-chip communication, and memory access. For computation approximation, the computation effort is reduced (e.g., using low precision adder/multiplier [2], loop perforation [10], etc.) to either accelerate execution or reduce power consumption. Loop perforation [6] transforms loops to skip selected instructions or iterations. Loop truncation [28] drops last few iterations of the loop. In loop memoization [29], for some iterations inside a loop, the results are computed and cached, for other iterations, the previously cached results are used instead of computing.

For on-chip communication, approximate NoCs drop data packets before they are injected into the network and recover them at the destination to reduce network workloads. [19] allows data to be compressed using pattern based NoC compression techniques, by compressing similar patterns into a shorter vector to reduce the volume of data movement across the chip. [5] drops data according to runtime buffer utilization of each router, and adaptively controls the traffic injection rate online with low hardware overhead.

Accessing memory values can also be approximated, e.g., reducing DRAM refresh rate [12], storing/accessing data in a compressed format [13], and speculating on the results of loads [3]. For a cache miss, load value approximation avoids remote cache accesses by predicting the value locally.

Since all the above-mentioned works focus on approximating one particular component, without coordinated optimization cross different layers, they might miss the opportunity to explore a large search space to find global optimal solutions. A motivational experiment is shown in Fig.1 to show the potential performance optimization by approximating computation with multiple approximate knobs simultaneously (e.g., loop truncation and data dropping).

Fig.1 shows the performance and output accuracy by simultaneously tuning three knobs, i.e., loop truncation, data

dropping, and computation precision scaling in an exhaustive search in benchmark swaptions. In loop truncation, iterations of error-tolerant loops in the application source code are skipped according to the quality requirement of the application [10]. The loop truncation rate can be 0%, 20%, 40%, 60%, and 80% of the total loop iterations. In data dropping, each network interface (NI) is equipped with a data dropper and a recoverer, for data dropping and recovering, respectively. Data packets are selectively dropped before they are injected into the network. The packet drop rate knob is set to be 0%, 20%, 40%, 60% and 80% of the total packets. In precision scaling, inaccurate multipliers are used instead of normal multipliers. The approximate mantissas of floating-point numbers are 0, 4, 8, 16 bits respectively. Three schemes use loop truncation, data dropping, and computation precision scaling separately are compared to the approach which optimizes them simultaneously in terms of performance and accuracy. The output of the benchmark swaptions is a list of prices, and the error is the average relative error between the approximate output price and the non-approximate one.

As shown in Fig.1, the optimization approach with the three knobs searches a much larger solution space and thus finds the Pareto frontier of performance (execution time) and output error. In contrast, the other three schemes search a very limited solution space and thus fail to find the optimal solutions. This observation motivates us to optimize multiple approximate knobs across layers simultaneously. In this paper, the error model, performance model, and power model are first built with respect to the loop truncation, data dropping, and computation precision scaling. An optimization problem is defined to maximize performance under the quality and power constraints. This problem has a large search space. Therefore, an efficient algorithm is proposed to solve the problem with low complexity such that the final error is within a user-defined bound.

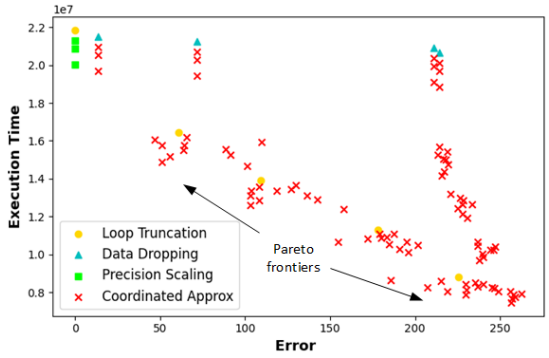

Fig. 1. Execution time vs. error for benchmark swaptions, when the four approaches are applied.

The rest of this paper is organized as follows. Related work is reviewed in Section 2. In Section 3, three approximation knobs are introduced. In Section 4, the error model, performance model, and power model are defined, the followed by the optimization problem definition. This problem is solved by an efficient algorithm in Section 5. Experimental results are evaluated in Section 6. Finally, the paper concludes in Section 7.

II. Related Work

Approximate computing [8][9], as an emerging computation paradigm, trades application output quality for energy-efficiency in applications with error forgiveness. Approximate computing approaches span from application layer to circuit layer.

(1) Approximation in the application layer. Power consumption is reduced using simplified functions in programs, e.g., loop perforation [10], precision scaling [14], using program versions of different accuracies [30]. In [10], critical loops that are not fault-tolerant are filtered out to identify adjustable loops, followed by a perforation space exploration algorithm which perforates all combinations of tunable loops to find pareto-optimal perforation choice. Domain specific knowledge is leveraged for approximations in applications and their algorithms. For machine learning applications, AxNN [16] proposes a method to transform any given neural network into an approximate neural network. This is performed by adapting the backpropagation technique to quantify the impact of approximating each neuron on the overall network quality and selectively approximating those neurons that impact on network quality. For iterative numerical computations applications, researchers proposed approximation techniques. In [31], ApproxIt presents a lightweight quality estimator that is able to capture the solution quality of each iteration and guide the selection of approximation mode in the next iteration, and it ensures the quality of final outputs at algorithm-level.

(2) Approximation in the system layer. Since the on-chip interconnection is an important component of a many-core system, recent studies try to drop data in NoCs. APPROX NoC [19] allows data to be compressed using existing pattern based NoC compression techniques, by compressing similar patterns into a shorter vector to reduce the volume of data movement across the chip. ABDTR [5] drops data in a fixed interval, while recovering the lost data based on linear interpolation. Packets are dropped according to the runtime buffer utilization of each router, which controls the traffic injection rate with low hardware overhead. ACDC [7] formulates a network performance optimization problem for approximate NoCs. Based on the flow prediction and application quality model, the controller first solves the congestion minimization problem, followed by exploiting the remaining error budget to reduce zero load latency.

(3) Approximation in the microarchitecture layer. The circuit design is simplified to achieve an approximate operation of the desired function, such as addition, multiplication and division. The approximate arithmetic units include approximate adder [25], approximate multipliers [33] and approximate dividers [18]. In [17], the number of transistors/gates of hardware modules are reduced and these modules are replaced by using less complex approximate counterparts. Approximate full adders were proposed in [25], where lower significant bits are approximately computed and upper significant bits are accurately computed. In [4], a novel error-tolerant adder (ETA) was proposed that improves performance and power consumption by eliminating or curtailing the carry propagation.

(4) Approximation in the circuit layer. Circuits, adjusted to operate at extreme conditions, can achieve lower power

consumption, with techniques including voltage over-scaling [26] and frequency over-scaling. There are some other works that study the logic synthesis for approximate computing circuits [15], using gate elimination, logic restructuring, gate pruning, timing speculation, and they automatically synthesize approximate circuits ranging from arithmetic building blocks to entire data paths.

III. The Approximate Knobs and System Architecture

In this section, the system architecture for approximate computing with three approximate knobs, loop truncation, data dropping, and precision scaling is introduced.

A. Overview of the System Architrcture

The manycore system is connected by NoC, where each node is composed by a core, an L1/L2 cache, a router and a network interface. Many applications include the error-tolerant loop codes that tradeoff performance and error by reducing the number of iterations (e.g., convergence loops in iterative solvers). In NoC, each network interface (NI) is equipped with a data dropper and a recover. A data dropper drops part of data in a packet according to the given drop rate, before the packet is injected into the network. After receiving such an incomplete packet, the data recoverer recovers the missing data, according to the received data. Each core has an accurate floating-point multiplier and a set of inaccurate floating-point multiplier, according to the given knob setting, the multiplexer selects the corresponding floating-point multiplier (see Section 3.4).

B. Loop truncation

The error-tolerant loop is selected by analyzing the application source code. For example, swaptions is a financial analysis application which uses a Monte Carlo simulation to solve a partial differential equation that prices a portfolio of swaptions. The trade-off between error and performance can be achieved by reducing the number of iterations of the loop which implements Monte Carlo simulation (In function *HJM_Swaption_Blocking*)

The original loop is shown in Algorithm 3.1, *Code Segment A* is the code in the iteration. The loop truncation works as in Algorithm 3.2. According to the given loop truncation rate x_1 , the last $n \times x_1$ iterations in the loop are skipped. A variable is used to compare the iterator index i with $n \times x_1$, where n is the total number of iterations of the loop. If $i > n \times x_1$, this iteration will be skipped. Otherwise, the iteration is executed. Therefore, the number of iterations to be executed is $n \times (1 - x_1)$.

Algorithm 3.1: Original Loop

```
for (int i=0; i < n; i++) {
    Code Segment A
}
```

Algorithm 3.2: Loop after Perforation

```
Input: x1 : loop truncation rate
for (int i=0; i < n × (1 − x1) ; i++) {
    Code Segment A
}
```

The range of the three approximation knobs are as follows. To avoid completely unacceptable results and program crashes, the maximum perforation rate for the error-tolerant loop is set to be 90% rather than 100%. The loop truncation knob $x_1 \in X_1 = \{\beta_1, \dots, \beta_p\}$, where $\beta_i \in [0,0.9]$, $\beta_i < \beta_{i+1}$. The iterations are skipped based on the loop truncation rate $\alpha_i \in X_1$, which means the number of iterations after skipping is $(1 - x_1)\%$ of the original iterations.

C. Aproximate NoC

Each NI has a data drop and a recoverer. The NI of the source node receives data from the processor and compresses the data by dropping a percentage of data units according to data drop rate, followed by packetizating the compressed data [5]. At the destination NI, the data is decompressed to a given data with the same length of the original (not compressed) ones. The data payload of a packet is deemed as a vector of data units (e.g., a 32-bit integer), denoted as a_1, a_2,.. The process of data dropping is similar to sampling, which skips one data unit after scanning every l data units. The skipping interval l is determined by the data drop rate x_2. The value of the skipping interval l is encapsulated together with other meta-data in the header flit.

After receiving a lossy packet at the destination, the payload (a_{l+1}, a_{2l+2}) is recovered based on the skipping interval l. For a skipped bit a_i, the corresponding recovered data unit a_i in the destination node is the average of its neighboring units.

$$l = \lfloor 1/x_2 \rfloor, x_2 \in (0, 0.5) \quad (1)$$

$$a_i = (a_{i-1} + a_{i+1})/2 \quad (2)$$

For a special case that a bit has only one neighbor (the MSB or LSB), it is recovered to be the value of its neighbor.

The data dropping knob $x_2 \in X_2 = \{\alpha_1, \dots, \alpha_n\}$, where $\alpha_j \in [0,0.5]$, $\alpha_j < \alpha_{j+1}$, $j = 1, \dots, n$. A drop rate $\alpha_j \in X_2$ indicates that the amount of data after dropping is $(1 - x_2)\%$ of the original data. When the data unit is dropped every l interval in the packet, the data drop rate reaches a maximum of 50% (when the interval l is 1).

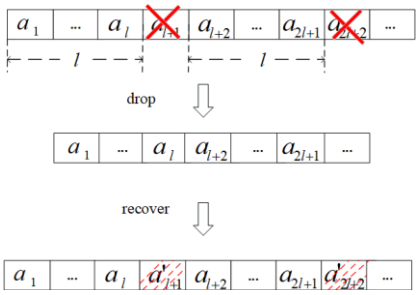

Fig. 2. Data dropping and recover.

D. Precision Scaling

Floating point number adopts IEEE754 standard, and inaccurate multiplier approximates the mantissa part (single-precision floating point number includes 23-bit mantissa). To support precision scaling, the processor has an accurate 32 bits multiplier, and a few inaccurate floating-point multipliers

MMBS [33], which are implemented for $N_1, N_2, \dots, N_n$ valid mantissa bits respectively, and $N_i \in [0,23]$.

As shown in Fig. 3., n-way multiplexer MUX is used to select the multiplier. Before the program runs, we set Sel and select a multiplier for the program multiplication calculation. The multiplier is selected according to the value of Sel, MUX selects the first multiplier when $Sel = 00$, selects the second multiplier when $Sel = 01$, selects the third multiplier when $Sel = 10$, and so on. The first multiplier is accurate, and the last multiplier is least accurate. x_3 is the precision scaling knob, which selects a multiplier. The precision scaling knob $x_3 \in X_3 = \{\gamma_1, \dots, \gamma_d\}$, where γ_k is an integer, $\gamma_k < \gamma_{k+1}$, $k = 1, \dots, d$.

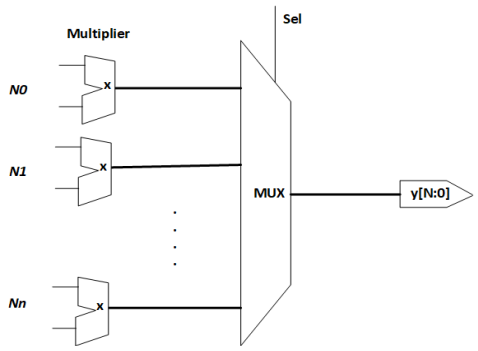

Fig. 3. The set of multipliers .

IV. Models and Problem Definition

A. The error, performance, and power models

The error, performance and power model are developed according to these three knobs as follows. The input parameters of the three models are loop truncation knob x_1, data drop knob x_2, and precision x_3. The outputs of the three models are application error, application execution time, and application average power, respectively. The trained data for the models are collected as follows, the loop truncation knob, data dropping knob, and precision scaling knob are randomly selected from X_1, X_2, and X_3. The output of the applications, application execution time, and application average power are generated.

The application error is obtained by comparing the approximate and non-approximate outputs. The error statistics are defined by the error measures provided by users for different applications, including Mean Relative Error (MRE), Mean Absolute Error (MAE) etc. (see Table2 in section 6). For example, the output of the benchmark blackscholes is a list of prices, and the error is the average relative error between the approximate output price and the non-approximate price.

The error, performance and power model can be fitted using polynomial regression model $h(\cdot)$ of order τ, $f(\cdot)$ of order δ and $p(\cdot)$ of order σ, respectively.

$$h(x_1, x_2, x_3) = \sum_{i,j,k=0}^{\tau} \phi_{ijk} x_1^i x_2^j x_3^k \quad (3)$$

$$f(x_1, x_2, x_3) = \sum_{i,j,k=0}^{\delta} \varphi_{ijk} x_1^i x_2^j x_3^k \quad (4)$$

$$p(x_1, x_2, x_3) = \sum_{i,j,k=0}^{\sigma} \omega_{ijk} x_1^i x_2^j x_3^k \quad (5)$$

where x_1, x_2, x_3 are the approximate control knobs. ϕ_{ijk}, φ_{ijl}, ω_{ijl} are the regression coefficients and ϕ_{000}, φ_{000}, ω_{000} are constant. Based on training data, the polynomial models are fitted using the gradient descent method with L1 and L2 regulations [21].

B. Problem Definition

The performance optimization problem is to optimize the overall performance of the system under the constraints such that the error and power should be below their respect thresholds. The problem can be described as follows: given an application, a user defines the program error threshold θ and power budget P_{in}, minimize the program execution time by selecting x_1, x_2, x_3 ($x_1 \in X_1$, $x_2 \in X_2$, $x_3 \in X_3$), while satisfying the quality and power requirements. It is formulated as:

$$Minimize\ f(x_1, x_2, x_3) \quad (6)$$

$$Subject\ to\ g_1(x_1, x_2, x_3) = \theta - h(x_1, x_2, x_3) \geq 0 \quad (7)$$

$$Subject\ to\ g_2(x_1, x_2, x_3) = P_{in} - p(x_1, x_2, x_3) \geq 0,$$

$$x_1 \in X_1, x_2 \in X_2, x_3 = X_3 \quad (8)$$

V. The Proposed Method

A. The Optimization Algorithm

An efficient algorithm is proposed to solve the problem in Eqn. 6-8. Before the applications are launched, the algorithm is run to find the optimal approximate.

Exhaustive search is the simplest method of knob selection. However, it is only suitable for a few knobs and a limited number of settings [32]. Since the original problem in section 4 has a relatively large number of settings, we try to solve the problem using convex optimization theory.

The original problem in Section 4 is a nonlinear programming problem. The interior point method [22][23][24] is adopted to solve it, which adds a logarithmic barrier with a penalty factor upon the original objective function to construct a dual problem. Let a vector X to represent the variable x_1, x_2, x_3, where $X = (x_1, x_2, x_3)^T$. The dual problem is formulated as:

$$Minimize\ \xi(X, t) = f(X) - \frac{1}{t}\sum_{i=1}^{2} ln[g_i(X)] \quad (9)$$

where $\xi(X, t)$ is the penalty function, $f(X)$ is the performance model in Eqn. 5, $g_1(X)$, $g_2(X)$ are the inequality constraints in Eqn. 7 and Eqn. 8. t is the penalty factor and as t approaches ∞, the penalty function $\xi(X, t)$ is closer to the original objective function $f(X)$. The logarithmic barrier function $ln[g_i(X)]$ is part of penalty function $\xi(X, t)$. According to the mathematical definition of logarithm, $g_i(X) \geq 0$, which makes the penalty function $\xi(X, t)$ satisfy the inequality constraint of Eqn. 7 and Eqn. 8. $2/t$ is the duality gap, which is the difference between the original solution and the dual solution.

Algorithm 5.1: Interior Point Optimization

Input: $f(X)$: The original objective function

$g_1(X)$: Inequality constraint in Eqn. 7
$g_2(X)$: Inequality constraint in Eqn. 8
Output: X^* : the optimal solution
1: Initialize initial feasible solution $X^{(0)} = (0,0,0)^T$
2: Initialize the number of inequality constraints $m = 2$
3: Initialize initial penalty factor t_0
4: Initialize penalty factor increasing coefficient μ
5: Initialize accuracy level ε
6: $t = t_0$
7: **repeat**
8: construct penalty function $\xi(X, t)$
9: Starting from $X^{(0)}$ use Newton's method [27] to find the extreme point $X^*(t)$ of $\xi(X, t)$
10: $X^{(0)} = X^*(t)$
11: $t = \mu t$
12: **until** $m/t \leq \varepsilon$
13: **return** $X^*(t)$

Algorithm 5.1 describes the iterative steps of the interior point method. The algorithm inputs are the objective function $f(X)$, inequality constraints $g_1(X)$ and $g_2(X)$. The output is the optimal solution X^*. There are three hyper-parameters, i.e., the initial penalty factor t, the penalty factor increasing coefficient μ, and the accuracy level ε. Each outer iteration (line 6-11) defines a new unconstrained objective function $\xi(X, t)$ (line 7). Next, Newton's method [27] is used to solve the unconstrained objective function $\xi(X, t)$, and obtains the optimization result $X^*(t)$ (line 8). Newton's method is an iterative algorithm, which approaches the optimal value gradually as the number of iterations increase. Finally, the starting point $X^{(0)}$ of penalty function for the next iteration and the penalty factor t are updated (line 9-10). The iterative process is repeated until the duality gap m/t is less than ε (line 11), and $X^*(t)$ as the solution to the algorithm. The time complexity of the outer iteration is $O(log\ 1/\varepsilon)$ with fast speed of convergence.

VI. Experimental Evaluation

A. Experimental Setup

In this section, we compare the proposed approximate method with three approximate methods, (1) loop truncation [28], which identifies redundant sub-computations, and reduces additional loop iterations of the application, (2) ABDTR [5], which drops data at fixed intervals, and recovers the lost data based on linear interpolation, and (3) the inaccurate multiplier MMBS [33], which divides the mantissa into a short exactly processed part and the remaining approximately processed part, and uses a new addition and shifting method to the approximate part to replace the multiplication. Experiments were evaluated using a cycle-accurate many-core simulator [1], with the system configurations listed in Table 1.

We evaluate our methodology using 5 applications from PARSEC [11] and AxBench [20]. These 5 benchmarks cover various important domains including data mining, financial analysis, media processing, computer vision, physics simulation, etc. The quality metrics and quality requirement (QR) settings of these applications are listed in Table 3 [7]. Two quality requirement settings are used for evaluation, with QR1 being strict and QR2 loose.

TABLE I. System Configuration

Numberof processors		64(Alpha 21264 ISA)	
Fetch/Decode/Commit size		4/4/4	
L1 D cache (private)		16KB, two-way, 32B line, two cycles, two ports, dual tags	
L1 cache (private)		32KB, two-way, 64B line, two cycles	
L2 cache (shared)		64KB slice/node, 128B line, six cycles, two ports	
Main memory size		2 GB, latency 200 cycles	
NoC flit size	32 bits	Meta packet	2 flits
NoC buffer	5×12 flits	NoC VC number	2
Data packet	32 flits	Routing algorithm	XY routing
NoC latency		Two cycles for router, one cycle for link	

TABLE II. Knobs Value Range

Loop truncation knob	[0, 0.9]
Data dropping knob	[0, 0.5]
Precision scaling knob	[0, 0.9]

TABLE III. Benchmark Configuration

Application	**Quality Metric**	**QR1**	**QR2**
rrrrrBlackscholes	Average relative error of output prices	0.2	0.4
Swaption	Average relative error of the output prices	0.25	0.5
Canneal	Relative error of the final routing cost	0.01	0.02
Fluidanimate	Percentage of particles not in same cell as in original results	0.3	0.7
Sobel	Average relative error of the output images	15	30

B. *Parameter Selection*

For the above algorithm, we have to choose the initial penalty factor t_0 and penalty factor increasing coefficient μ. μ and t_0 strikes a balance between outer iteration number and iteration number of each Newton's method. If μ or t_0 is too small, many outer iterations might be needed. If μ or t_0 is too large, the Newton's method takes long time to converge. The following experiments were designed to find the best parameters that makes the algorithm converge faster.

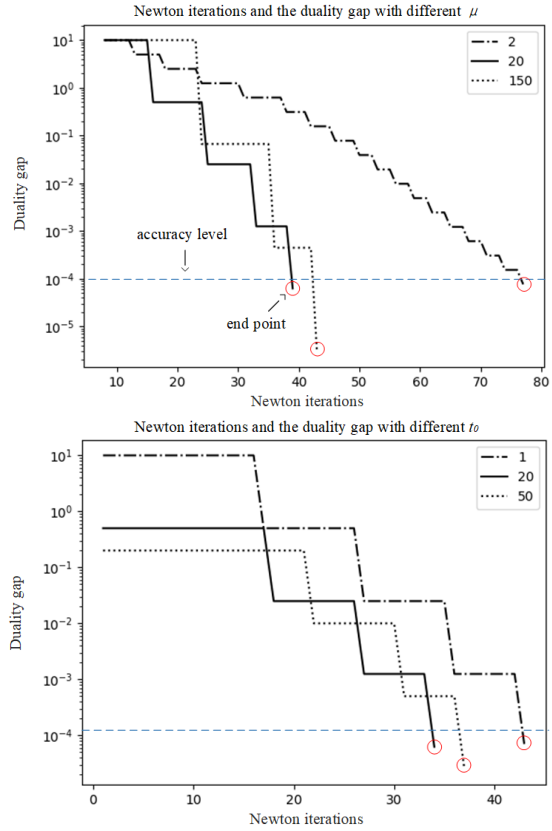

Fig. 4. The relation between the number of Newton iterations and the duality gap with different μ and t_0.

We use random numbers to generate a 3-dimensional, $m = 2$ inequality constrained quadratic programming problem of the same size as Section 4. The interior point algorithm is used to solve the problem, with accuracy level $\varepsilon = 10^{-4}$. Fig. 4 illustrates the relation between the number of Newton iterations and the duality gap with different μ and t_0. It can be seen from the Fig. 4, the convergence line of the algorithm shows a step-down trend. Each inflection point represents the update of the outer iteration, and the length of the horizontal line segment represents the number of iterations of Newton's method in the outer iteration. $\mu = 20$ and $t_0 = 20$ are best in terms of convergence speed in Fig. 4, which are used in the following experiments.

C. *Validation of the Error, Performance, and Power Models*

Fig. 5 show R^2 score [21] of the error, performance and power models for all the applications. An R^2 score close to 1 indicates lower regression error. On average, the R^2 scores of the error, performance, and power models are 0.9812, 0.9810, and 0.979, respectively. Therefore, the error, performance, and power models are fairly accurate.

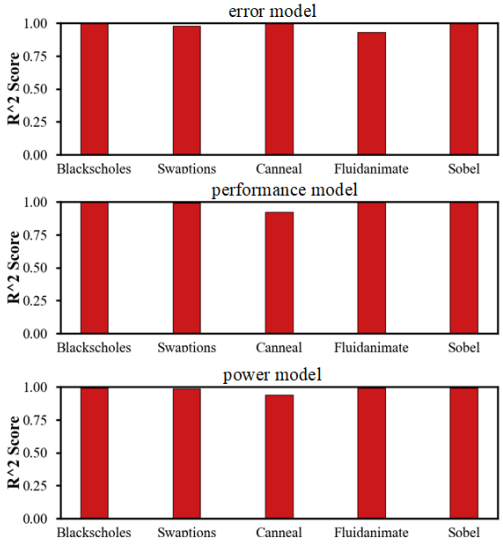

Fig. 5. R^2 score of the models.

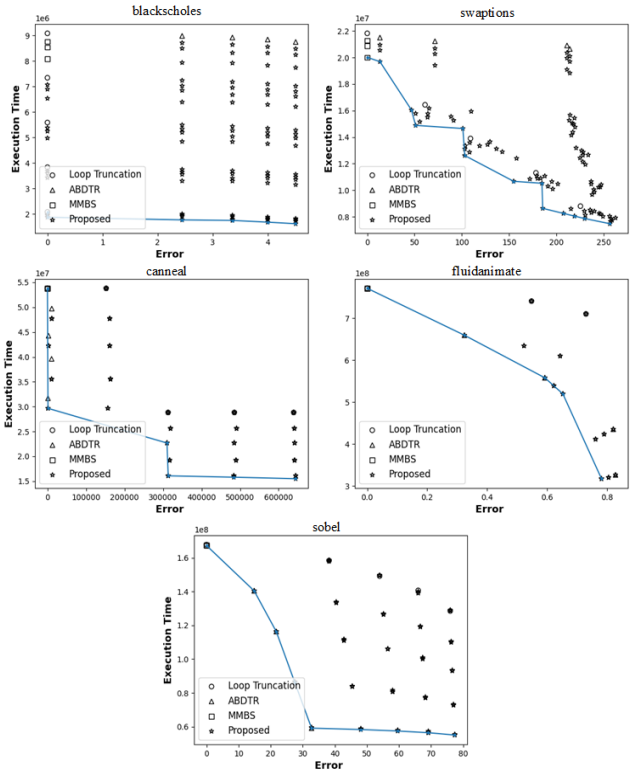

Fig. 6. Error versus execution time for the benchmarks.

D. Performance and Power Consumption Evaluation

Fig. 6 shows the execution time and error of the proposed scheme and those three schemes, loop truncation, ABDTR, and MMBS in benchmarks from PARSEC. The points in Fig. 6 shows the execution times and errors for different schemes. The blue points (connected by a line) from the Pareto-frontier, which is found by our proposed scheme, as it searches a much larger solution space and thus finds the Pareto frontier. In contrast, the other three schemes cannot find the Pareto frontier, which leads to either lower performance or high error.

Fig.7 compares the proposed method with loop truncation, ABDTR, and MMBS in terms of execution time and power consumption. When the quality requirement is QR1, on average, the proposed approximation accelerates execution by 25.42% over loop truncation, and accelerates execution by 32.49% over ABDTR, and accelerates execution by 41.40% over MMBS. For QR2, on average, the proposed approximation accelerates execution by 43.05%, 38.70%, and 62.89%, compared with loop truncation, ABDTR, and MMBS. For some applications (e.g., Canneal), the proposed approximation achieves as much as 54.46% execution speedup over these three approximate methods.

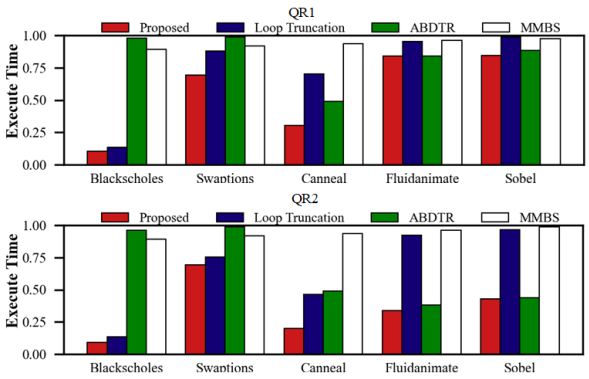

Fig. 7. Execution time of benchmark with different approximation methods under different quality constraints QR1 and QR2.

The reason is as follows. Using the loop truncation knob for computation intensive application is more beneficial as reducing the number of iterations can effectively reduce the computational workload. On the other hand, to reduce communication latency, using the data dropping knob for a communication intensive application is more beneficial as reducing data transmission on NoC can alleviate network congestion. The proposed scheme considers both computation and communication, thus the performance is better than single knob approximate methods.

Finally, Fig. 8 shows the power consumption under quality requirements QR1 and QR2. For QR1, on average, the proposed scheme reduces power consumption by 33.86%, 31.04%, and 44.90% over loop truncation, ABDTR, and MMBS, respectively. For QR2, on average, the proposed scheme reduces power consumption by 60.88%, 37.68%, and 69.65% over loop truncation, ABDTR, and MMBS, respectively.

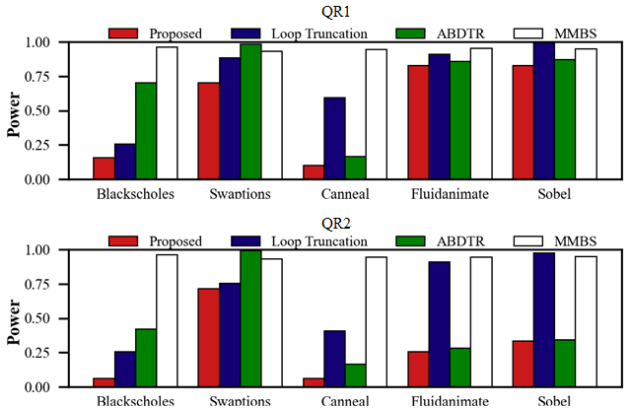

Fig. 8. Power consumption of benchmark with different approximation methods under different quality constraints QR1 and QR2.

E. Overhead

The proposed algorithm is implemented as software codes and run offline. The average execution time of the is about cycles, which is found to be 1.29% of the benchmark execution times, on average.

According to the synthesis result reported by Synopsys Design Compiler and PrimePower targeting a $45nm$ TSMC library, the NoC approximation method cost $968um^2$ and 0.636 mW on area and power, respectively. Using DSENT [34] with a $45nm$ CMOS technology, a router with the configurations in Table 1 consumes $145mW$ power, and its area is $462{,}965um^2$. The area and power consumption of the data dropper and recoverer are 0.20% and 0.44% of the router, respectively. The 4-way multiplexer cost $18um^2$ and $0.0012mW$. The accurate multiplier cost $594.8um^2$ and $0.02mW$. The area of the three inaccurate floating-point multipliers is 282.9, 253.0, and 192.3 um^2, and the power is 0.00879, 0.00734, and $0.00551mW$. As a comparison, a CPU core [35] consumes $0.846W$ and its area is $2mm^2$. The area and power consumption of the multiplier approximation are 0.066% and 0.005% of the core.

VII. Conclusion

In this paper, we proposed an approximate method for many-core systems. Three different approximate knobs were set in the applications and system to change the approximation level. The performance and power consumption optimization problem for approximate many-core systems was formulated. To solve this problem, a lightweight algorithm was proposed. An application error model, performance model, and power model were built, and based on the models, the lightweight algorithm selected the optimal approximate knobs combination. Experimental evaluation confirmed that the proposed approximate method significantly outperforms the other single-knob approximate methods in terms of both performance and power consumption.

Acknowledgment

This research program was supported by the National Natural Science Foundation of China 61971200, Open Research Grant of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of

Sciences CARCH201916, Zhejiang Lab 2021LE0AB01, and National Key Research and Development Program of China under Grant 2019QY0705.

REFERENCES

[1] X. Wang, M. Yang, Y. Jiang, P. Liu, M. Daneshtalab, M. Palesi, and T. Mak, "On self-tuning networks-on-chip for dynamic network-flow dominance adaptation," ACM Trans. Embedded Comput. Syst., vol. 13, no. 2s, pp. 73:1–21, 2014.

[2] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in Proc. IEEE/ACM Int. Conf. Computer-Aided. Design, 2013, pp. 48-54.

[3] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in Proc. IEEE/ACM Int. Symp. Microarchitecture, 2014, pp. 127–139.

[4] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo and Z. H. Kong, "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing," IEEE Trans. VLSI. Syst., vol. 18, no. 8, pp. 1225-1229, 2010

[5] L. Wang, X. Wang and Y. Wang, "ABDTR: Approximation-Based Dynamic Traffic Regulation for Networks-on-Chip Systems," in Proc. ICCD, 2017, pp. 153-160.

[6] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation," in Proc. Int. Conf. Supercomputing, 2018, pp. 341-351.

[7] S. Xiao, X. Wang, M. Palesi, A. K. Singh and T. Mak, "ACDC: An Accuracy- and Congestion-aware Dynamic Traffic Control Method for Networks-on-Chip," in Proc. DATE, 2019, pp. 630-633.

[8] Q. Xu, T. Mytkowicz, and N.S. Kim, "Approximate computing: A survey," IEEE Design & Test, vol. 33, no. 1, pp. 8–22, Feb. 2016.

[9] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in Proc. DAC, 2015, pp. 1-6.

[10] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. "Managing performance vs. accuracy trade-offs with loop perforation," in Proc. ACMSIGSOFT Symp. European Conf. Foundations. Software Engineering, 2011, pp. 124–134.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in Proc. Int. Conf. Parallel Archit. Compilation Techniques, 2008, pp. 72–81.

[12] A. Raha, S. Sutar, H. Jayakumar and V. Raghunathan, "Quality configurable approximate DRAM," IEEE Trans. Computers., vol. 66, no. 7, pp. 1172–1187, July 2017.

[13] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan, "Approximate memory compression for energy-efficiency," in Proc. ISLPED, July 2017, pp. 1–6.

[14] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in Proc. DAC, 2013, pp. 113:1–9.

[15] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: Systematic logic synthesis of approximate circuits," in Proc. DAC, 2012, pp. 796–801.

[16] S. Venkataramani, A. Ranjan, K. Roy, and A. Roy. "AxNN: energy-efficient neuro-morphic systems using approximate computing," in Proc. ISLPED, 2014, pp. 27–32.

[17] H. A. F. Almurib, T. N. Kumar, F. Lombardi, "Inexact designs for approximate low power addition by cell replacement," in Proc. DATE, 2016. p. 660–665.

[18] L. Chen, J. Han, W. Liu, and F. Lombardi, "Design of approximate unsigned integer non-restoring divider for inexact computing," in Proc. GLSVLSI, 2015, pp. 51–56.

[19] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. j. Kim, "APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures," in Proc. ISCA, 2017, pp. 666-677.

[20] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," IEEE Design & Test, vol. 34, no. 2, pp. 60-68, April 2017.

[21] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning: data mining, inference, and prediction," Springer Science & Business Media, 2009.

[22] R. H. Byrd, J. C. Gilbert, and J. Nocedal, "A Trust Region Method Based on Interior Point Techniques for Nonlinear Programming," Math. Program., vol 89, no. 1, pp. 149–185, 2000.

[23] R. H. Byrd, M. E. Hribar, and J. Nocedal, "An Interior Point Algorithm for Large-Scale Nonlinear Programming," SIAM J. Optim., vol 9, no. 4, pp. 877–900, 1999.

[24] R. A. Waltz, J. L. Morales, J. Nocedal, and D. Orban, "An interior algorithm for nonlinear optimization that combines line search and trust region steps," Math. Program., vol 107, no. 3, pp. 391–408, 2006.

[25] M. Ramasamy, G. Narmadha and S. Deivasigamani, "Carry based approximate full adder for low power approximate computing," in Proc. ICSCC, 2019, pp. 1-4.

[26] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in Proc. DATE, 2011, pp. 1–6.

[27] S. Bahlak, "Convex Optimization: Algorithms and Complexity," In Foundations and Trends in Machine Learning, vol. 8, no. 3-4, pp. 231-357, 2014.

[28] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in Proc. ICSE, 2010, pp. 25-34.

[29] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in Proc. ACM SIGSOFT Symp. European Conf. FSE, 2011, pp. 102-112.

[30] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variableaccuracy algorithms," in Proc. Int. Symp. CGO, 2011, pp. 85–96.

[31] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, "Approxit: An approximate computing framework for iterative methods," in Proc. DAC, 2014, pp. 1-6.

[32] X. Sui, A. Lenharth, D. S. Fussell and K. Pingali, "Proactive Control of Approximate Programs," ACM SIGPLAN Notices, vol. 51, no. 4, pp. 607-621, 2016.

[33] J. Li, Y. Guo, S. Kimura, "Accuracy-Configurable Low-Power Approximate Floating-Point Multiplier Based on Mantissa Bit Segmentation," in Proc. TENCON, 2020, pp. 1311-1316.

[34] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic, "DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in Proc. IEEE/ACM Int. Symp. Networks-on-Chip, 2012, pp. 201–210.

[35] Y. Yuyama, M. Ito, Y. Kiyoshige, Y. Nitta, S. Matsui, O. Nishii et al., "A 45nm 37.3GOPS/W heterogeneous multi-core SoC," in Proc. ISSCC, 2020, pp. 100-101.