Phase-Aware Optimization in Approximate Computing

Subrata Mitra[†], Manish K. Gupta[⋆], Sasa Misailovic[‡], Saurabh Bagchi[†]
[†]Purdue University, USA [⋆]Microsoft, USA [‡]University of Illinois, Urbana-Champaign, USA

Abstract

This paper shows that many applications exhibit *execution-phase-specific* sensitivity towards approximation of the internal subcomputations. Therefore, approximation in certain phases can be more beneficial than others. Further, this paper presents OPPROX, a novel system for application's execution-phase-aware approximation. For a user provided error budget and target input parameters, OPPROX identifies different program phases and searches for profitable approximation settings for each phase of the application execution. Our evaluation with five benchmarks and four existing transformations show that our phase-aware optimization on average does 14% less work for a 5% error tolerance bound and 42% less work for a 20% tolerance bound.

Keywords Execution Phases, Approximate Computing

1. Introduction

Approximate computing trades accuracy of computation for savings in execution time and/or energy by leveraging approximation opportunities across the computing stack, including programming languages[14, 34, 38, 40], compilers[7, 26, 42, 43], runtime systems[9, 18, 21], and hardware[17, 19, 32, 41]. In addition to data-parallel and streaming applications[23, 24, 39] researchers proposed approximation techniques suitable for iterative numerical computations, such as iterative solvers, large scale numerical models, and sparse matrix calculations[16, 46, 47].

Approximate computing techniques typically introduce inexactness and/or approximation by transforming compute-intensive kernels, which we call *approximable blocks* (ABs). Furthermore, many approximation techniques expose *knobs* to calibrate the approximation levels (ALs), which control the error or speedup. For instance, *loop perforation* [26, 43] skips a fraction of a loop's iterations, and exposes this fraction as a knob to control the accuracy/speedup tradeoff.

Outer-Loop Pattern. Many applications follow a computation pattern in which the majority of computation is performed inside a main loop (we refer to it as the *outer loop*) encompassing all the ABs. Examples of outer loops include *timestep* loops in scientific simulations, *convergence* loops in iterative solvers, or *enumerator* loops for processing a series of information blocks (e.g., video frames).

For large applications with multiple ABs, the trade-off between speedup and error becomes complex. Often the optimum configuration of ALs in the various ABs are not obvious, especially if the approximation of internal ABs influences the number of iterations of the outer loop. Off-line exhaustive search can be possible only for a small number of approximate program configurations[43], and the majority of the previous approaches used various heuristic search strategies based on representative inputs[7, 14, 26, 38], or dynamically tuned the ALs based on the observed errors from the approximated regions[7, 25, 38, 44]. While many of these techniques identify and leverage properties of specific code patterns in ABs, they typically apply the same transformation for the entire execution and/or input. Such fixed optimization choices may lead to rigid transformed programs that miss fine-grained optimization opportunities.

Phase-Aware Optimization. For many iterative computations, the outer loop controls the precision of the final solution. Here, the iterations of the outer loop naturally segment the overall application execution into multiple *phases*. We define a phase as *a segment of execution that has distinct speedup or error characteristics*. For example, a numerical solver execution can go through an initialization phase, a maturity phase, and a convergence phase.

Our experimental results show that two different phases of the computation may generate different amounts of error for the same level of approximation. This exposes a new opportunity for optimization algorithms – they can select not just *how much* to approximate, but also *in which phase* to approximate. We find empirically that for some applications (such as LULESH[3]), approximating one phase may induce almost 8X less error than applying the same approximation in another phase of the execution.

Our Solution Approach. We present OPPROX, a system for phase-aware optimization of approximate programs. OPPROX takes as inputs: a program with tunable approximable blocks and a user-provided *accuracy specification*, which consists of (1) a set of representative inputs that exercise the application's desired functionality, (2) an accuracy metric that tells how to compute the difference between the results of the exact and the approximate execution, and (3) an error budget e_b that specifies how much reduction in the accuracy metric in the final output the user is ready to tolerate.

OPPROX operates in four conceptual steps. First, OPPROX identifies different computation phases. Second, OPPROX models the speedup and error generated due to different levels of approximation in the individual ABs *and* in different computation phases using representative inputs. Third, OPPROX compares the benefits of various approximation settings in different phases and splits the overall error budget e_b into phase-specific error budgets in proportion to the predicted benefits. Finally, OPPROX formulates phase-specific trade-off space

exploration as a numerical optimization problem and finds the most profitable approximation settings for each phase using the phase-specific error budgets as the constraints.

We show that for many applications, *both* the approximation level and the phase in which approximation is performed, have significant contributions towards the final error. Hence, phase-specific optimal approximation settings can provide good speedup (which we express here using the number of instructions executed) even under constrained error budget. When compared to an oracle but phase-agnostic version from prior works[43, 44], our approach on average provides 42% speedup compared to 37% from the oracle version for an error budget of 20% and for a small error budget of 5% provides on average 14% speedup compared to only 2% achieved by the phase-agnostic oracle version.

Contributions. This paper makes following contributions:

1. We introduce the concept of phase-specific approximation for controlling the approximation error and improve application speedup.
2. We introduce modeling of application speedup and approximation error based on polynomial regression that captures the dependency on input parameters and the phase of the computation.
3. We define the phase-specific approximation space exploration as a numerical optimization problem and present an algorithm to find profitable configurations for multiple approximations under a given error budget.
4. We evaluate OPPROX on five benchmarks and four existing approximations. The results show that phase-aware approximation becomes very attractive for improving speedup (especially when operating under low error budget) compared to phase-agnostic approximation from prior works[43, 44].

2. Example

We explain the motivation for OPPROX and how it works using LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) as an example. LULESH[3] is a widely used hydrodynamic application that simulates the Sedov blast wave problem[37] in three dimensions. It represents a typical hydrodynamics code that solves the hydrodynamics equations by partitioning the equations spatially. Fig.1 gives an abstract representation of the main computation part in LULESH. In the main computation, LULESH iterates through an outer-loop until the simulation reaches a stable state (*i.e.*, until a condition called the *Courant condition* is met). Inside the outer-loop, it computes several physical quantities. At the end, LULESH reports the final energy for each of the elements it simulated.

Accuracy Specification. The quality metric for LULESH is the difference in the final energy from the approximate run compared to that from the accurate execution and averaged across all the elements.

Application Profiling. Given an user-provide error budget, OPPROX finds the best configuration to maximize the speedup of LULESH. We represent the speedup in terms of the number of instructions executed in the program. At first, we profile LULESH to find the 6 most compute intensive kernels (lines 5-10 in Fig.1). Then, OPPROX applies three approximation techniques – loop perforation, loop truncation, and memoization (discussed in Sec. 3.2).

```
1  elements = A set of elements to simulate
2  While(state->stable ==false)
3  {
4      increment_simulation_time();
5      forces_on_elements();
6      acceleration_of_elements();
7      velocity_of_elements();
8      position_of_elements();
9      strain _of_elements();
10     calculate_timeconstraints();
11     state = get_current_state();
   }
```

Figure 1: Abstract computation pattern for LULESH. The `while` loop iterates until the simulation achieves stable state.

Ultimately, we find approximation in four of these logical functions: forces_on_elements, position_of_elements, strain_of_elements and calculate_timeconstraints, did not lead to either a hang or crash or unusable quality of service (QoS) degradation. These four kernels form the approximable blocks (ABs) for LULESH. This process of finding ABs is analogous to the one in [26]. The approximation levels (ALs) corresponding to these kernels were exposed as environment variables and each one can be set to different levels from 0 to 5 with 0 being the accurate run and 5 being the run with the maximum approximation.

Phase-Specific Behavior of Approximable Blocks. For some ABs, as we increase the approximation level, as expected, we observe application speedup as well as an increase in QoS degradation, as shown in Fig.2. However, while running LULESH with different combinations of ALs corresponding to different ABs, we observed that the number of iterations in the outer-loop also varies significantly as can be seen in Fig.3 — when run without any approximation the outer-loop iterates 921 times, with some combinations of ALs it increases to 965 times and as a result *slows down* the application instead of speeding it up. The maximum speedup we observed was 1.34 but at a cost of 38% QoS degradation. Further, we explored whether approximating only during some selected duration of the execution would help us to have a better control over QoS degradation and speedup. In this example, we divided the outer-loop iterations into 4 phases—each phase comprises an equal number of outer-loop iterations— and selectively approximated only in one phase. We show the results in Fig. 4 and Fig. 5, where the different points within one phase correspond to different combinations of the ALs. We see that approximating in phase-1 can provide speedup but also drastically degrades the QoS. This behavior can be explained from the intrinsic nature of the algorithms involved in LULESH. Approximation during the initial phases makes it difficult to meet the stable condition leading to QoS degradation. Approximation in later phases reduces the impact of such errors as the accurate execution of the first phase already took the simulation much closer to the golden values. For example, approximation only in phase-4, generates negligible error but still can provide some speedup

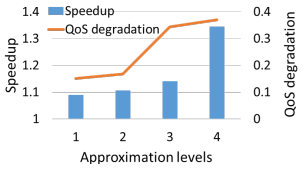


Figure 2: Both speedup and error increase with approximation levels of the blocks in LULESH.

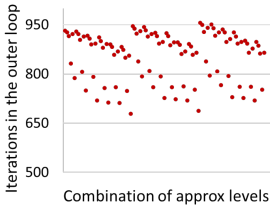


Figure 3: Variation in the number of iterations made by the outer-loop in LULESH.

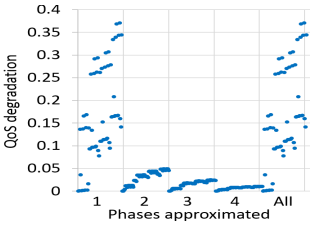


Figure 4: LULESH phase-specific QoS

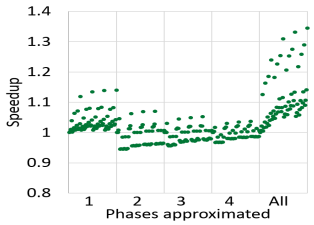


Figure 5: LULESH phase specific speedup

with some approximation settings. Therefore, phase-specific approximation gives better opportunity to find a suitable combination for speeding up the application, *especially when operating under low error budget*.

Phase-Aware Optimization of Approximate Blocks. OPPROX takes several steps to build such phase specific speedup and QoS degradation model for LULESH as illustrated in Fig.6. At first, OPPROX instruments the LULESH code by adding log messages to capture the call-context corresponding to the ABs. Then LULESH is run with different combinations of its input parameters (length of cube mesh and number of regions) and the sequence of unique call-contexts for the ABs are extracted from the logs. These sequences are used to classify control flows based on input parameters and build separate models for each distinct control flows to capture input-parameter-dependent behavior.

For LULESH, OPPROX automatically divides the execution into 4 phases. For each phase, it builds polynomial regression models for speedup and QoS degradation by using random samples of different combinations of input parameter combinations and ALs for that phase. For LULESH, the R^2 score corresponding to the prediction of the final QoS degradation and the speedup were 0.94 and 0.99 respectively.

Application-level Optimization. Now, for a user-provided QoS budget, OPPROX uses optimization to find the best phase specific ALs. At first, it allocates sub-budget (a portion of the QoS degradation budget) to each phase in proportion to the mean value of the ratios of how much speedup is gained from that phase to the amount of QoS degradation. Then OPPROX finds the best combination of approximation for that phase subject to the allocated sub-budget. Any unused sub-budget from one phase is reallocated to the other phases. For LULESH, initially sub-budgets allocated to the 4 phases are in proportion to 0.166, 0.17, 0.265, and 0.399 of the full budget e_b. Using the optimized settings suggested by OPPROX, for error budgets (e_b) of 20%, 10%, and 5%, the approximate versions of LULESH achieved speedups of 1.28, 1.21, and 1.17 respectively.

3. Opprox

Fig.6 presents OPPROX's workflow. OPPROX performs an offline training using execution logs collected from multiple runs of the application on the training inputs (Sec.3.3). During the training, OPPROX builds two sets of models. The first set of models predicts the control flow of the application (including the number of loop iterations) by taking as input

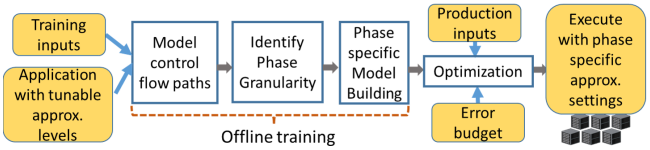


Figure 6: Workflow of OPPROX

the application's input parameters. To build these models, OPPROX uses decision-tree classification (Sec. 3.4). The second set of models predicts the speedup and QoS degradation of application by taking as inputs (1) the control flow from the first set, (2) the application's input parameters, and (3) the possible approximation levels for each AB. To build these models, OPPROX uses polynomial regression (Sec.3.6). At runtime, OPPROX finds phase-specific approximations for a user-provided QoS degradation budget. Specifically, it uses the offline models to formulate and solve a numerical optimization problem that aims to maximize the speedup while keeping the QoS degradation within the budget (Sec.3.8).

3.1 Opprox Inputs

Application With Tunable Approximation Levels. OPPROX relies on the user to identify the ABs and implement suitable approximation techniques that provides tunable approximation levels. In general, for compute intensive kernels the computation time decreases (*i.e.* speedup increases) with an increase in the AL. At the same time, the QoS degrades with an increase in the AL due to the inaccuracies introduced by the technique, as illustrated in Fig.2 for LULESH. To choose the ABs, we follow the sensitivity profiling procedure presented in prior work [26]. In particular, these techniques filter out the blocks where approximation makes the program to crash or results in unacceptable-quality output. As part of the sensitivity profiling, we try different approximation techniques on the compute intensive blocks and finally choose a set of blacks that are both compute intensive and can withstand certain levels of approximation.

QoS Metric. The Quality of Service (QoS) is an application specific metric that captures how different are the results from approximate computing when compared to results produced by an *exact* computation, and denote it as δQoS. Some applications have common domain-specific metrics, e.g., for image or video processing applications the QoS can be the value of Peak Signal to Noise Ratio (PSNR). For the applications that do not have a domain-specific QoS metric, we use a default *distortion* [34], which computes the relative

scaled difference between the outputs generated from an approximate computation and an exact computation.

3.2 Examples of Approximation Techniques

There are many available techniques [17, 32, 43] that can be used to approximate an AB. The concept of OPPROX is generic and can be applied with *any* approximation technique that provides multiple ALs for each AB. Here, we assume that the approximation exposes the variable `approx_level`, which controls the approximation level for each of the techniques. In this paper, we analyze four previously proposed techniques:

Loop perforation: In loop perforation [26, 43], the computation time is reduced by skipping some iterations, as shown below. The behavior essentially samples the result space.

```
for (i = 0; i < n; i = i + approx_level)
    result = compute_result();
```

Loop truncation: In this technique [26, 43], we simply drop last few iterations as shown in the following example:

```
for (i = 0; i < (n − approx_level); i++)
    result = compute_result();
```

Memoization: In this technique [13], for some iterations inside a loop we compute the result and cache it. For other iterations we use the previously cached results.

```
for (i = 0; i < n; i++)
    if (0 == i % approx_level)
        cached_result = result = compute_result();
    else result = cached_result;
```

Parameter tuning: In some applications, there exist some input parameter which can directly be used to control the accuracy of the computation [21]. For example, in Bodytrack, the parameters `min-particles` and the *number of annealing layers* is suitable for this purpose. Overall, users can provide a list of the parameter names and the set of values that the parameter may take.

Algorithm 1 Finding proper phase granularity

```
1: app ← Application under test
2: thresh ← Phase sensitivity threshold for QoS
3: N ← 2 # Number of phases
4: maxDiffPrev = getMaxQoSDiff(app,N)
5: while True do
6:     newN=N · 2
7:     maxDiffNew = getMaxQoSDif(app,newN)
8:     if abs(maxDiffPrev - maxDiffNew) > thresh then
9:         N ← newN
10:    else
11:        Break
12:    end if
13: end while
```

3.3 Sampling for Training Data

OPPROX collects training data by profiling the instrumented application with different combinations of ALs for each AB and a variety of representative input parameters provided by the user. During each run it collects the call-context logs *i.e.*, the sequence of ABs executed, the number of instructions executed by each AB in each iteration of the outer-loop, and the QoS degradation w.r.t the golden output obtained from corresponding accurate execution. OPPROX first builds local models for each AB, hence for each AB, it exhaustively covers the corresponding AL-space, while executing all other ABs accurately. Then, to capture the interaction due to approximations in multiple ABs, random sparse samples are collected where approximation levels in *all* the ABs are arbitrarily set between zero (accurate computation) and the maximum approximation level. We assume the number of discrete ALs in each AB are not high (usually between 4-8, in our case), hence to build good local models, exhaustive sampling is required. In case, number of discrete ALs are high, sparse sampling can also be used for the local models. The training data consists of such sampling for each phase.

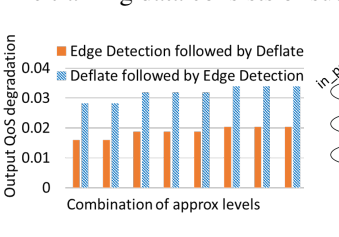


Figure 7: FFmpeg: Changing the order of two filters: Deflate and Edge Detection, results in significant change in the QoS degradation.

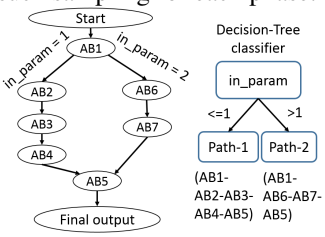


Figure 8: OPPROX uses decision-trees to predict input-parameter dependent control-flow variations.

3.4 Predicting Control-Flows

Application's control-flow, *i.e.*, the sequence of executed ABs may change depending on the input-parameters resulting in different speedup and QoS degradation characteristics. For example, Fig. 8 presents a case in which, depending on the value of the input parameter `in_param`, the application can execute two different control-flow paths (i.e., two different sequences of ABs). Fig.7 shows that swapping the order of two ABs in FFmpeg (*edge detection* and *deflate* filters) drastically changes the QoS degradation.

OPPROX trains a *decision tree* classifier model [30] using the call-context logs that are obtained from executing the program on the training inputs. The model can predict the control-flow that the application takes for a given combination of input parameters. Later, for for each unique control-flow of the application, OPPROX creates separate predictive models for speedup and QoS degradation (Sec. 3.6). An ideal training-set should capture all the distinct control-flow paths that can be encountered in the production.

3.5 Identifying Phase Granularity

The QoS degradation of a variety of applications depends not only on the ALs in the ABs but also on the phase of application's execution in which the approximation was done. Before constructing the models, OPPROX first determines the number of logical phases N that the application's execution should be divided into. While, in general the entire execution of the application might perform distinct set of tasks such as initialization, warming-up, computation, epilogue etc., OPPROX only focuses on the main computation inside the main outer-loop[1].

[1] We identify such loops by manually inspecting the source code. However, automatic identification is also possible, *e.g.*, selecting the most compute intensive loop using the loop-profiling feature of Intel C++ compiler.

Algo. 1 presents how OPPROX searches for the appropriate number of phases. First, it divides the total number of iterations (I) in the outer-loop into $N = 2$ equal-sized phases, each consisting of approximately I/N loop iterations[2]. Second, the algorithm executes the application multiple times, applying different approximation settings within one phase at a time, and measures the speedup and QoS degradation at the end of each execution. The helper `getMaxQoSDiff` function (1) runs the application with N phases and multiple approximation settings and (2) calculates the maximum difference between the mean QoS degradations of any two executions, which correspond to approximations in consecutive phases. Third, the algorithm continues increasing the number of phases while the difference between the QoS degradations between the consecutive phases is above the user-provided threshold (Algo. 1, lines 8-12).

While a large number of phases N can capture the relationship between the phases and the QoS degradation or the speedup at finer level, it would increase the size of the search space (and the training time) exponentially. The user-specified threshold for the difference in results between two consecutive number of phases helps bound this value of N.

3.6 Performance and Error Models

Estimating Iteration Counts. We define the speedup in terms of the computation or amount of work done, *i.e.*, the total number of instructions executed as follows:

$$S = \frac{\#(\text{instructions executed in accurate run})}{\#(\text{instructions executed in approximate run})}$$

As a result of approximation, each AB gets some speedup. However, the final speedup of an application depends not only on the local speedups gained at the ABs, but also the number of iterations of the outer loops encompassing those ABs. Recall, the number of iterations of these outer-loops can be constant (*e.g.*, in a predefined timestep based simulation), input parameter dependent (*e.g.*, in FFmpeg, depends on the number of frames in a video) or can depend on internal approximations (*e.g.*, in LULESH). OPPROX extracts the number of iterations of the outer-loop by calculating how many times a call-context sequence of ABs has repeated in the execution log. OPPROX builds estimators for the number of outer-loop iterations using polynomial regression [29], which has the approximation settings of the internal ABs and input parameters as the modeling input.

Modeling Speedup and QoS Degradation. Approximation levels at the ABs and the corresponding phases dictate how much speedup can be obtained and how much QoS degradation would be incurred. For each unique control-flow path, OPPROX builds separate models to capture the phase-specific application speedup and QoS degradation, using a two-step approach.

The first step builds local models, which capture the speedup or QoS degradation when only one of the ABs is approximated. Thus, each AB has a local speedup and QoS degradation model per phase. These models take as input the ALs for the particular AB and input-parameter combination provided to the application. For example, for a approximable-block AB_1 with a_1 as the knob for AL and input-parameter combination P, the local speedup model for AB_1 is captured as: $s_1(a_1, P)$.

The second step builds models to capture the combined effect on overall speedup or QoS degradation when multiple ABs are approximated simultaneously. These models use the prediction (speedup or QoS degradation) from the models from the first step as inputs. For example, for an application with two approximable-blocks AB_1 and AB_2 with local speedup models s_1 and s_2 respectively, the overall speedup model is captured as: $S(s_1, s_2)$. As previously mentioned, OPPROX uses polynomial regression to build both local and overall models. For example, the previously mentioned overall speedup S would be modeled in a degree-2 polynomial regression as: $c_0 + c_1 s_1 + c_2 s_2 + c_3 s_1 s_2 + c_4 {s_1}^2 + c_5 {s_2}^2$, where the coefficients c_0,...,c_5 are computed by the regression algorithm using the training data.

The final QoS degradation or speedup also explicitly depends on how many times each AB is called *i.e.*, how many times the outer-loop executes. This is because, as for the QoS, the more number of times an AB is called, the more QoS degradation it is likely to create due to approximation error. For speedup, it is slightly more complex, an approximation of an inner AB may actually increase the number of iterations of the outer-loop and thus *decrease* the speedup. Let us take an example of an outer-loop with only one AB inside. Let W be the amount of work done by this block per iteration. After approximation the amount of work performed by this block becomes w resulting in a local speedup of: x=W/w. Say, at the same time the number of iteration of the outer-loop changed from I to i. due to approximation. Then, the overall speedup of the application would be: $(I \cdot W)/(i \cdot w)$ or $I \cdot x/i$. Thus the overall speedup would depend not only on x but also on the change in the number of iterations in the outer-loop i. Hence, we first use polynomial regression to build a highly accurate estimator for outer-loop iterations and then explicitly use the estimated value as an input feature while building our overall speedup and QoS models to ensure at least one input variable will closely dictate the output.

Confidence Analysis of Models. There might be errors in these machine learning based models itself because the training data does not exhaustively capture all the possible scenarios due to combinatorial explosion. To address this problem, OPPROX calculates a confidence interval of its models by adapting the approach from [28]. For example, if OPPROX predicts Q as the QoS degradation value for a particular approximation combination and p fraction of the time modeling error remains within $e\%$, then it interprets that actual QoS degradation for that approximation settings can be anywhere between $[Q - e, Q + e]$ which form the confidence interval. Here p is a means of controlling the *confidence*

[2] When I is not divisible by N, the remainder is added to the final phase.

in the prediction. To remain conservative, OPPROX use the upper limit of the p=0.99 confidence interval as the effective QoS degradation and use the lower limit in case of speedup estimation. This ensures that we avoid the risk of going over the QoS degradation budget.

3.7 Improving Modeling Accuracy

To reduce noise during polynomial regression based modeling, OPPROX uses Maximal Information Coefficient (MIC) [33], to determine if an association exists between any given input feature, which could be an input parameter to the application or the approximation level of any AB, and the target output of the model, which could be the number of iterations of the outer loop, the degradation of QoS, or the speedup of the AB. Features not having an association are filtered out.

After this filtering, OPPROX gradually increases the degree of the polynomial regression until it finds a good R^2 score with 10-fold cross validation. In 10-fold cross-validation, as per standard practice, the original training data is randomly partitioned into 10 equal size subsets. Of the 10 subsets, 9 subsets are used for training and the remaining single subset is used for testing the model. This process is then repeated 10 times, with each of the 10 subsets used exactly once as the test data. The 10 results from the folds are then averaged to produce a final estimation. Cross-validation avoids overfitting.

OPPROX checks if the models achieve a target accuracy (*i.e.*, a good R^2 score). The value of this target accuracy is a design choice, *e.g.*, a value greater than 0.9 may denote a good model. If OPPROX finds that the models are not accurate enough for the entire input data set, it breaks the input into smaller subcategories and attempts to build a model for each subcategory. To create the sub-models, OPPROX splits the values of a feature put in magnitude order into k subsets, and learns separate models for each subset.

3.8 Optimization Framework

The final goal of OPPROX is to find the optimal settings for the ALs for each phase of the application that would maximize the speedup of the application, for a given QoS degradation budget QoS_b specified by the user. The overall optimization algorithm framework is shown in Algo. 2.

Phase Specific Allocation of QoS Degradation. By analyzing various applications we found that the same approximation in different phases of the execution achieves different speedups, and causes different levels of QoS degradation. For the purpose of our optimization we define a metric called *return on investment* (ROI) of QoS degradation budget for a phase ph as follows:

$$roi_{ph} = \frac{1}{m}\sum_{i=1}^{m}\frac{S_i}{\delta QoS_i} \quad (1)$$

Here, m is the number of available training data points for the phase ph. S_i is the speedup for the i_{th} data point and δQoS_i is the corresponding QoS degradation. Intuitively, the ROI value for a phase gives a statistical measure of how much benefit we are likely to get at the expense of certain amount of QoS degradation for that phase.

OPPROX divides the overall QoS degradation budget across all the phases of execution *in proportion* to their corresponding ROI values. Thus, for a given a QoS degradation budget QoS_b, the share of the budget allocated to the phase ph would be: $normROI_{ph} \cdot QoS_b$, where $normROI_{ph}$ is the ROI of this phase normalized by the sum of the ROIs of all the phases. This is a policy decision of how to divide the overall QoS degradation and OPPROX can accommodate other policies than the one described above. OPPROX searches the configuration space among the phases in the decreasing order of their ROI values and any QoS budget left-overs are redistributed among the remaining phases.

Algorithm 2 Finding phase specific approximation settings

```
1: QoS_b ← Total QoS degradation budget
2: models ← Phase specific approximation models
3: sortedPhases ← sortPhasesBasedOnROI()
4: for all (phase in sortedPhases) do
5:     normROI ← calculateNormalizedROI()
6:     phaseQoSBudget ← QoS_b · normROI
7:     phaseModel ← models[phase]
8:     consumedQoS = optimizePhase(phaseModel, phaseQoSBudget)
9:     QoS_b ← QoS_b - consumedQoS
10: end for
```

Finding the Optimal Settings for Each Phase. OPPROX uses the QoS degradation budget allocated to each phase to find the optimum settings for approximation for that phase that would maximize the speedup.

Assume there are M, ABs and $A = (A_1, \ldots, A_M)$ denotes the *configuration* of the ALs for these blocks for a phase ph. The values each A_i can take are the discrete approximation levels for the corresponding block. Let $S(A)$ be the speedup of the application, and $\delta QoS(A)$ be the QoS degradation as a result of these approximations. Thus, OPPROX's goal is to find the optimum value of A for phase ph that will maximize the speedup while keeping the overall QoS degradation within the budget:

$$\begin{aligned} \underset{A}{\text{maximize}} \quad & S(A) \\ \text{subject to} \quad & \delta QoS(A) \leq normROI_{ph} \cdot QoS_b \end{aligned}$$

OPPROX estimates the value of S and δQoS for each A using the models previously described in Sec. 3.6 and solves a (polynomial) numerical optimization problem. This step is represented as the function `optimizePhase` in Algo.2.

4. Experimental Methodology

For evaluation, we use five representative applications and benchmarks from a wide variety of domains. Here, we describe these applications and implementation of OPPROX.

4.1 Description of the Applications

LULESH: We provided details for LULESH in Sec. 2.

CoMD: CoMD[1] is a representative application for a broad class of molecular dynamics(MD) simulations. In general, the method of MD simulation involves the evaluation of the force acting on each atom due to all other atoms in the system

Apps	Input parameters	Approx. techniques used	Search space (# approx. settings)
LULESH	length of cube mesh, # regions	loop perforation, loop truncate, memoization	699,840
FFmpeg	frames per second, video duration, bitrate, filters	loop perforation, memoization	207,360
Bodytrack	# annealing layers, # particle, # frames	loop perforation, input-tuning	1,966,080
PSO	Swarm size, dimension	loop perforation, memoization	14,400
CoMD	# unit cells, lattice parameter, # timestep	loop perforation, loop truncate	229,500

Table 1: Application specific input parameters, approximation techniques used and number of combinations explored

and the numerical integration of the Newtonian equations of motion for each of those atoms.

QoS Metric: At the end of the simulation, the energy of the system is expressed in terms of the potential and kinetic energy of the atoms. As the QoS metric, we use the difference in potential and kinetic energy compared to the accurate execution and averaged across all the atoms.

Computation Pattern: CoMD's main computation is surrounded by an outer loop which iterates for the number of simulation timesteps provided as the input. This outer loop internally calls several compute intensive functions. CoMD outer loop represents a classic timestep loop in scientific computations where the number of timesteps is an input parameter. The outer loop iteration for CoMD does not depend on any other input parameters or the ALs of the internal ABs.

FFmpeg: FFmpeg[2] is a widely used video processing toolkit which provides a large number of filters to process a video, like edge detection filter, blur, color balance, deshake etc. These can be combined in various ways for a specific type of processing.

QoS Metric: We use PSNR (peak signal to noise ratio).

Computation Pattern: FFmpeg passes encoded video to a decoder which produces uncompressed frames (raw video). Inside an outer loop, FFmpeg applies a series of filters on each frame to process the video in various ways. After filtering, the frames are re-encoded and passed to a multiplexer, which writes the encoded packets to the output file. FFmpeg outer loop represents typical streaming analytics loops. The number of iterations depends on the input parameter, the number of video frames, and not on the ALs.

Bodytrack: Bodytrack[8] is a computer vision application that uses an annealed particle filter and videos from multiple cameras to track the movement of a human through a scene.

QoS Metric: QoS metric is the distortion of the vectors that represent the position of the body parts. The weight of each vector component is proportional to its magnitude. Vector components which represent larger body components therefore have a larger influence on the QoS metric than vectors that represent smaller body components .

Computation Pattern: For every frame of the input videos, the application extracts the image features and computes the likelihood of a given pose in a annealed particle filter. The main computation is inside an outer convergence loop. Inside the loop, the likelihood weight for each particle is calculated and if that results in an invalid model, particles are removed. Bodytrack's outer loop is also a type of convergence loop. The number of iterations depend on the number of annealing layers and not on the internal ALs. However, when the value of `min-particles` is small, the iteration count also depends on the ALs.

Particle Swarm Optimization: Particle swarm optimization (PSO)[22] is a population-based stochastic approach for solving continuous and discrete optimization problems. We used an implementation for continuous functions (called the objective functions). PSO has similarity to evolutionary computation where the algorithm is expected to move the swarm of particles toward the best solutions.

QoS Metric: It is the average difference of the values of the best fitness vector calculated for each particle in the swarm.

Computation Pattern: PSO starts with a population of candidate solutions, also called particles. The main computation is inside an outer-loop which iteratively improves a candidate solution until the convergence criterion is met. In each iteration, the computation computes new positions and velocities of the particles in the search space.

4.2 Implementation Details

In this section we briefly discuss some of the design choices and implementation details. Table 1 summarizes the total number of approximation combinations we collected. We had 4 ABs for LULESH and Bodytrack, and 3 ABs for CoMD, PSO and FFmpeg. Depending on the application and the AB, we used between 4 to 8 different approximation levels and up to 27 different input combinations. While trying to find optimal number of phases for dividing the application execution, we explored up to N=8 phases. For regression models, we found the polynomial degrees varied between 2 to 6 for all the models in our applications corresponding to an R^2 score greater than 0.9.

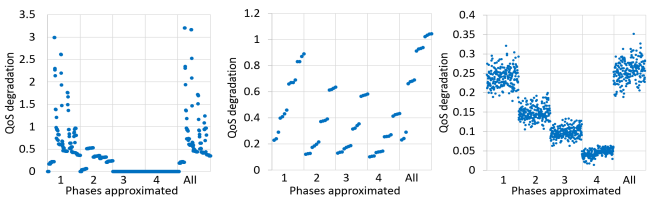
What happens at the runtime. For each application, the trained models are stored as Python's serialized `pickle` format in designated locations. User submits the job with a target error budget in a configuration-file. Then a runtime-script loads the corresponding models and finds the best phase-specific approximation settings for that error budget using OPPROX's optimizer on the trained-models and invokes the `SLURM` native scheduler. The phase-specific approximation settings are passed to the job via environment variables; specifying the approximation level for each AB during each phase of the execution.

5. Evaluation

We now present the experimental results. We performed all evaluations on 64-bit Intel Xeon Phi machines with 64GB of RAM running RHEL 6.6, 64-bit OS. Following the same approach as [21, 25, 43, 44], we ran all the applications in serial mode using one thread. Applications were compiled with `gcc` version 4.8.4 and with `O3` optimization.

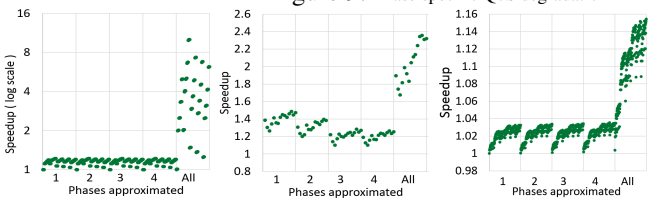
5.1 Phase Specific Behavior

First, we show how the QoS degradation and speedup varies as we turn on approximation in different phases. To show a



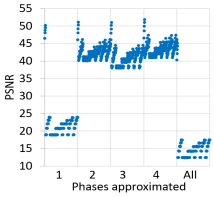
(a) CoMD (b) PSO (c) Bodytrack

Figure 9: Phase specific QoS degradation

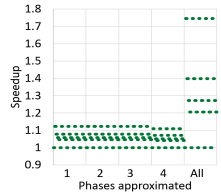


(a) CoMD (in log scale) (b) PSO (c) Bodytrack

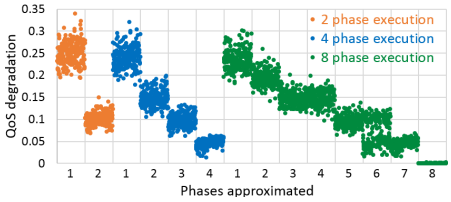
Figure 10: Phase specific speedup



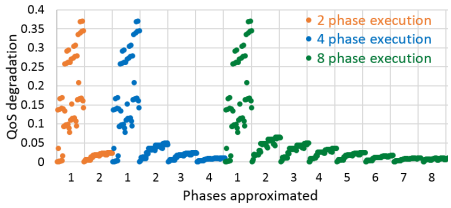
(d) FFmpeg (Y-axis is PSNR, higher is better)



(d) FFmpeg



(a) Bodytrack



(b) LULESH

Figure 11: Characteristics of QoS degradation for execution divided in to 2, 4, and 8 phases

visually comparable phase-specific behavior, for all the applications we divide the main computation into 4 phases of equal length. Here a phase is defined in terms of the number of iterations in the main outer loop. Fig.9 presents QoS degradation and Fig.10 presents the speedup characteristics resulting from different combination of approximation levels in the ABs. Corresponding results for LULESH is in Fig.4 and Fig.5. Each point in the plots represents a distinct approximation setting, *i.e.*, a different configuration of ABs. The X-axis is divided in segments showing the QoS degradation and speedup characteristics when approximations were applied only to that phase, letting all other phases run accurately. The last segment (marked as "*All*") shows the behavior when approximation was turned on for the entire duration of the application execution. For all the applications except FFmpeg, Y-axis shows the percentage of degradation in QoS (lower is better). For FFmpeg, Y-axis is the value of PSNR (Peak Signal to Noise Ratio) and a higher value represents lower approximation error. The Y-axis in Fig.10a for CoMD speedup is in log scale.

5.1.1 Error Characterization

For all the applications we studied, approximation in the first phase introduces maximum approximation error resulting in significant QoS degradation. For LULESH and CoMD, error introduced in the first phase of the execution is so significant that its effect on QoS degradation is comparable to the execution where approximation is turned on for the entire duration. It can also be observed that as we turn on the approximation in the later execution phases, its impact on QoS degradation diminishes. Approximation during the fourth phase of the execution creates almost insignificant QoS degradation. This behavior can be explained from the intrinsic nature of the algorithms involved in these applications as discussed in the context of LULESH (Sec. 2).

CoMD. Approximation during the initial phases puts the particles further from their accurate positions with vastly inaccurate values of kinetic and potential energies. Inaccurate positions and energy values create a ripple effect during the rest of the simulation and QoS degrades. The magnitude of the inaccuracies and the scope for their propagation is reduced if approximations starts only in the later phases.

PSO. PSO iteratively converges towards the best solution starting from a set of initial candidate solutions (particles). The quality of the solution set being explored in the current iteration depends on the accuracy of the solutions from the previous iterations. Hence, inaccuracies in the first few phases have significantly higher impact on QoS.

Bodytrack. Like for PSO, Bodytrack's QoS degradation is less affected if approximation is turned on at later phases.

FFmpeg. The outer loop iterates over the video frames applying multiple approximated filters on each frame. Although this application runs the same set of filters on each frame, the approximations in the first phase significantly reduce PSNR because the encoding procedure (which follows the filter execution) induces the dependency between the neighboring frames. For example, the second encoded frame only keeps the information relative to the first frame. Therefore, any error introduced in the first few frames propagated throughout the remaining frames (out of 150 frames in total) leading to a phase-dependent PSNR degradation.

5.1.2 Performance Characterization

From Fig.10, we see that phase-specific behavior for speedup have two distinct patterns. Either the speedup drops if we trigger approximation in later phases (*e.g.*, in Fig.5 for LULESH and in Fig.10b for PSO) or speedup remains almost unaffected with respect to which phase is being approximated (*e.g.*, for CoMD, Bodytrack, FFmpeg in Fig.10a, Fig.10c,and Fig.10d, respectively). Thus, for applications in the first category, it is most beneficial to approximate in the later phases, rather than uniformly, because speedup remains the same while QoS degradation is lower in the later phases.

5.1.3 Changing Phase Granularity

Fig.11 presents how changing the number of phases affects the QoS degradation for Bodytrack and LULESH as we

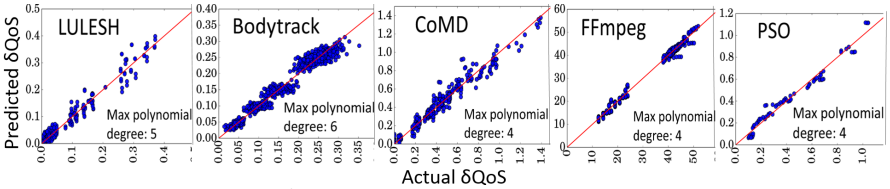


Figure 12: Prediction of QoS degradation

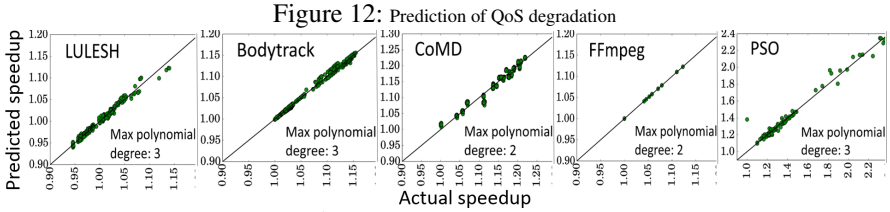


Figure 13: Prediction of speedup

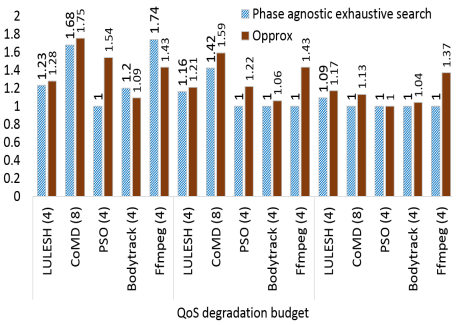


Figure 14: For different QoS budgets, comparison between OPPROX and phase-agnostic exhaustive search used by prior works [43, 44] as the idealized or oracle scheme.

uniformly divide the execution into 2, 4, and 8 phases. For both applications, when execution is divided into 2 phases, it is preferable to use aggressive approximation in phase-2 instead of phase-1 (especially when operating under low QoS degradation budget). The behavior is similar when execution is divided into 4 phases and it provides more fine granularity for controlling QoS degradation. However, when we divide the execution in 8 phases, the distinction between the QoS degradation coming from different phases becomes blurry – e.g., in the cases of Bodytrack (phase-3 and phase-4 have almost the same QoS degradation) and LULESH (phases 5 to 8). Thus, identifying proper phase granularity is important.

Fig.15a and Fig.15b present how phase specific QoS degradation and speedup characteristic varies for four different input parameter combinations (described in Sec. 4) for Bodytrack and LULESH, when the execution is divided into 4 phases. Each point represents a particular approximation setting for that phase and the color denotes the corresponding input parameter combinations. For both the applications, for all the four input combinations, we see a consistent trend in the behavior of QoS degradation and speedup with respect to various phase-specific approximations. This validates that the benefits of phase-aware approximation is not tied to any particular input parameter combination.

5.2 Evaluation of Modeling Accuracy

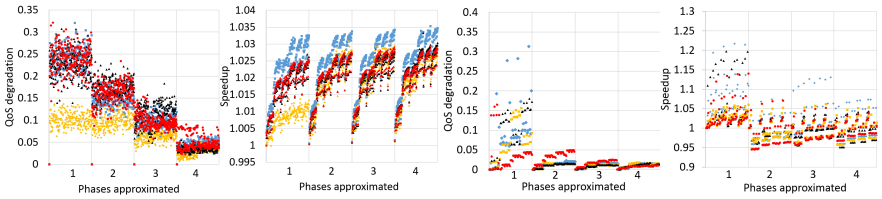
Fig.12 and Fig.13 show the result of the evaluation of the prediction accuracy of the models built by OPPROX. We randomly partitioned data into two equal-sized non-overlapping parts. The first part was used for training and other for testing. In the X-axis shows the actual value and the Y-axis shows the predicted value from our models. The diagonal line indicates a perfect prediction and deviations from this diagonal line indicates prediction error. For QoS degradation, as shown in Fig.12 OPPROX makes reasonably accurate predictions as bulk of the points are close to the diagonal line. QoS degradation for FFmpeg (which is PSNR) and PSO are highly predictable. However for LULESH, Bodytrack and CoMD, the model shows higher inaccuracies. Speedup models are also very accurate for all the applications as shown in Fig.13.

Overhead w.r.t phase granularity: Table 2 summarizes OPPROX's total training time and the time to find the phase-specific optimized approximation settings as we vary the phase granularity from 1 (i.e., phase-agnostic) to 8. The training is performed offline and is done only once, while optimization is done before scheduling the job with production inputs. The *optimization time* in Table 2 includes the total time for loading the models, optimizing, and scheduling the tasks. OPPROX's phase-specific optimization gives a trade-off in terms of the number of phases to use. Smaller number of phases incur lower overhead while higher number of phases give better control over speedup and approximation error. However, for applications that run for a long duration, such overheads become negligible compared to the benefits from the speedup. Moreover, most of the optimization overhead can be offset by precomputing the results for many common input parameters and error budgets.

5.3 Evaluation of Optimization Framework

To show the effectiveness of our proposed *phase-aware* optimization technique, we compare OPPROX with a *phase-agnostic* optimization through *exhaustive search*. Such phase-agnostic exhaustive search was used previously [43, 44] as an idealized oracle technique. Thus, essentially we compare OPPROX with the *best* achievable result by the phase-agnostic optimization. Phase-agnostic exhaustive search goes over *all* combinations of approximation settings to find which setting provides maximum possible speedup while keeping the corresponding QoS degradation within the budget. However, such phase-agnostic search does not consider any phase-specific approximations and applies the chosen approximation setting through the entire execution. Fig.14 presents evaluation using 3 levels of the QoS degradation budget: *large-budget* (20% degradation), *medium-budget* (10% degradation) and *small-budget* (5% degradation). Since for FFmpeg, QoS is calculated in terms of PSNR where a higher value signifies less error, we use target PSNR values of 10, 20, and 30 as the large-budget, medium-budget, and small-budget respectively. In Fig.14, within (), we mention the number of phases used.

Small Error Budget (5%). Phase-specific approximation improves performance for all benchmarks, while the baseline (phase-agnostic) approximation is unable to obtain any speedup in 4 of the 5 applications. For example, in case of FFmpeg, OPPROX achieved 37% speedup while phase-



(a) Bodytrack (b) LULESH

Figure 15: Phase specific characteristics of QoS degradation and speedup for different inputs. Each point represents an approximation setting. Points from different input combinations have different colors.

# Phases	1	2	4	8	1	2	4	8
Apps	Training time(sec)				Optimization time(sec)			
LULESH	274	281	379	1682	3.8	5.9	10.7	19.3
FFmpeg	297	307	497	2091	2.3	3.6	6.1	10.1
Bodytrack	373	471	1347	16038	6.9	13.2	22.1	41.7
PSO	165	173	291	3347	1.3	2.4	4.6	7.9
CoMD	226	271	419	5203	1.3	2.1	3.7	7.1

Table 2: Variation of OPPROX's training and optimization times w.r.t phase granularity.

agnostic search achieved nothing. as it could not find any approximation setting that would create lower than user-specified QoS degradation. On average, OPPROX gave 14% speedup while phase-agnostic search gave only 2%.

Medium Error Budget (10%). OPPROX improves performance for all benchmarks because OPPROX can perform search at a finer (phase) granularity, while phase-agnostic search was able to provide speedup only for LULESH and CoMD. Approximating CoMD during phase-3 and phase-4 creates insignificant QoS degradation (Fig.9a) but the share of speedup achieved is similar to phase-1 and phase-2 (Fig.10a). Thus, OPPROX can set a higher approximation level for phase-3 and phase-4 to increase speedup and choose a lower approximation level for phase-2 or phase-1 to keep the QoS degradation within budget.

Large Error Budget (20%). OPPROX can provide significant speedup (up to 75% for CoMD) for all the applications. However, for Bodytrack and FFmpeg, phase-agnostic search is able to find a better setting that gives higher speedup. For FFmpeg, the large budget is large enough to accommodate all possible approximation settings for the entire execution of the application. For Bodytrack, our model for QoS degradation computes a less precise prediction, which is a consequence of conservative confidence intervals that OPPROX computes around the predicted values.

These results jointly show that OPPROX can successfully use the concept of phase-specific approximation to fine-tune and control the error budget giving better speedup compared to phase-agnostic approximation method.

6. Related Work

We discuss related software-based approximations.

Software Systems for Approximation. Researchers have presented programming language support, including static analyses[11, 12, 27, 40, 48] and dynamic analyses[10, 26, 36] that quantify the effects of approximation. Researchers also proposed various (phase-agnostic) compiler transformations[13, 21, 26, 39, 43].

PetaBricks autotuner[5, 6], automatically finds configurations of alternative function implementations for a given QoS budget. An extension in [15] identifies classes of similar inputs using two-level clustering and applies different approximations for each input class. These are complementary to our approach as OPPROX can learn the control-flow of the input-optimized program generated by PetaBricks and then apply its phase-specific optimization.

Models for Input-Aware Optimization. In an early work, Rinard [34, 35] presented an approach that builds linear-regression models for various values of the accuracy knobs, but for individual inputs. More recently, Capri[44] constructs generalized models of performance and accuracy of the computation using M5 estimation algorithm[31]. The main difference between Capri and OPPROX is that they do not exploit the additional control coming from execution-phase-specific approximation levels. Laurenzano et al.[25] derive and runs canary inputs (smaller versions of the full inputs) to determine the approximation level based on input content, while focusing on streaming and data-parallel applications. In contrast, OPPROX uses input-parameters to predict control-flow variations that impacts performance and accuracy. OPPROX can also benefit from using canary inputs to more accurately model the phase-specific behaviors.

Runtime Systems and Middleware for Approximation. Existing adaptive techniques support on-line monitoring of accuracy[7] and latency[20, 21]. Approximation-aware runtime systems have also been used to improve resilience[4, 45], guide the execution of data-parallel applications[18, 38, 39], and reduce communication cost in parallel programs[9]. While adaptive mechanisms track program execution, they incur runtime overhead to dynamically build models and do not build specialized phase-aware models. In contrast, we build phase-aware models through offline training.

7. Conclusion

We introduce phase-aware approximation for finding the best approximation settings for different phases of the computation. We present OPPROX, a system that models speedup and QoS degradation corresponding to phase-specific approximation settings and maximize the speedup subject to an acceptable QoS degradation. OPPROX is compatible with many prior approximation techniques. Our evaluations show that, compared to oracle phase-agnostic baseline used by prior works, OPPROX can significantly improve performance, especially for tight QoS degradation budgets.

Acknowledgments

We thank Adrian Sampson and the anonymous reviewers for the valuable feedback. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-1629431, CNS-1527262 and CNS-1513197. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Classical molecular dynamics proxy application. `http://www.exmatex.org/comd.html`.

[2] Ffmpeg: Video processing framework. `https://ffmpeg.org/`.

[3] Lulesh: Livermore unstructured lagrangian explicit shock hydrodynamics. `https://codesign.llnl.gov/lulesh.php`.

[4] S. Achour and M. C. Rinard. Approximate computation with outlier detection in topaz. In *OOPSLA*, 2015.

[5] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.

[6] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for autotuning variable-accuracy algorithms. In *CGO*, 2011.

[7] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[9] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In *CGO*, 2015.

[10] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, 2010.

[11] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

[12] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[13] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *FSE*, 2011.

[14] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *DAC*, 2010.

[15] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *PLDI*, 2015.

[16] P. D. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. V. Palem, and T. Palmer. On the use of inexact, pruned hardware in atmospheric modelling. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2018), 2014.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[18] Í. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, 2015.

[19] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *ETS*, 2013.

[20] H. Hoffmann. Jouleguard: energy guarantees for approximate applications. In *SOSP*, 2015.

[21] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[22] J. Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. 2011.

[23] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: an online quality management system for approximate computing. In *ISCA*, 2015.

[24] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. Incapprox: A data analytics system for incremental approximate computing. In *WWW*, 2016.

[25] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input responsive approximation: Using canary inputs to dynamically steer software approximation. In *PLDI*, 2016.

[26] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *ICSE*, 2010.

[27] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, 2014.

[28] S. Mitra, G. Bronevetsky, S. Javagal, and S. Bagchi. Dealing with the unknown: Resilience to prediction errors. In *PACT*, 2015.

[29] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[30] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1 (1):81–106, 1986.

[31] J. R. Quinlan et al. Learning with continuous classes. In *Australian joint conference on artificial intelligence*, pages 343–348, 1992.

[32] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *DATE*, 2014.

[33] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti. Detecting novel associations in large data sets. *Science*, pages 1518–1524, 2011.

[34] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, 2006.

[35] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, 2007.

[36] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS*, 2015.

[37] M. Rogers. Analytic solutions for the blast-wave problem with an atmosphere of varying density. *The Astrophysical Journal*, 125:478, 1957.

[38] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *MICRO*, 2013.

[39] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

[40] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[41] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. *ACM TOCS*, 2014.

[42] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Wash-*

ington Technical Report UW-CSE-15-01, 2015.

[43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[44] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive control of approximate programs. In *ASPLOS*, 2016.

[45] V. Vassiliadis, K. Parasyris, C. Chalios, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos. A programming model and runtime system for significance-aware energy-efficient computing. In *PPoPP*, 2015.

[46] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. In *DAC*, 2014.

[47] Q. Zhang, Y. Tian, T. Wang, F. Yuan, and Q. Xu. Approxeigen: An approximate computing technique for large-scale eigen-decomposition. In *ICCAD*, 2015.

[48] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.