HPAC: Evaluating Approximate Computing Techniques on HPC OpenMP Applications

Konstantinos Parasyris, Giorgis Georgakoudis, Harshitha Menon, James Diffenderfer,
Ignacio Laguna, Daniel Osei-Kuffuor, Markus Schordan
{parasyris1,georgakoudis1,harshitha,diffenderfer2,ilaguna,oseikuffuor1,schordan1}@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA, USA

ABSTRACT

As we approach the limits of Moore’s law, researchers are exploring new paradigms for future high-performance computing (HPC) systems. Approximate computing has gained traction by promising to deliver substantial computing power. However, due to the stringent accuracy requirements of HPC scientific applications, the broad adoption of approximate computing methods in HPC requires an in-depth understanding of the application’s amenability to approximations.

We develop HPAC, a framework with compiler and runtime support for code annotation and transformation, and accuracy vs. performance trade-off analysis of OpenMP HPC applications. We use HPAC to perform an in-depth analysis of the effectiveness of approximate computing techniques when applied to HPC applications. The results reveal possible performance gains of approximation and its interplay with parallel execution. For instance, in the LULESH proxy application approximation provides substantial performance gains due to the reduction of memory accesses. However, in the leukocyte benchmark approximation induces load imbalance in the parallel execution and thus limiting the performance gains.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools; Runtime environments.**

KEYWORDS

Approximate Computing, Programming Models, Runtime Systems, OpenMP

1 INTRODUCTION

Even after the end of Dennard scaling, which stipulated a continuous increase in processor clock speeds by transistor miniaturization, the continuation of Moore’s law shifted processor designs to multi-core architectures for providing more computational power. However, as we approach the limits of Moore’s law, it has become necessary to explore new paradigms to shape future computer

architectures and software systems. A promising approach is *Approximate Computing* (AC), where computations trade-off accuracy for improved performance and energy efficiency.

In computer architecture, researchers have designed approximate hardware for CPUs [16, 35], memories [14], and accelerators [15, 18] to achieve better performance and energy efficiency. In addition to hardware-level approximation, there are significant opportunities for AC techniques in the software. Code and loop perforation approximation techniques [20] have shown up to 3× performance improvement with less than 10% loss of accuracy for image processing applications [20, 44]. Techniques such as approximate memoization, sampling, and tile approximations applied to stencil computation in image processing and financial software improve performance by at least 2× with small error [38]. Other examples include reduced precision techniques [37], which yields performance gains of up to 45%.

Approximate computing relies on the ability of the applications to tolerate some loss of accuracy in the computed output and trade that for better performance and energy efficiency. The idea of trading accuracy for performance is a common practice in the design of domain-specific algorithms for HPC applications—for example, using polynomial approximation [17] to functional forms, the inexact Newton’s method [10], and variable precision computing. However, gaining performance without affecting the accuracy requires significant, expert domain knowledge.

Non-HPC application domains, such as multimedia, image processing, and machine learning, adopt general software and hardware approximation techniques. These non-HPC workloads naturally tolerate some error with mild perceived effects in their outputs. However, in scientific computing application domains, even slight errors can destabilize computations and lead to spurious results. In general, scientific applications exhibit strict accuracy requirements, and the implementation of these applications is critical for life and physical sciences. Therefore, there is limited use of approximate computing techniques in HPC applications. As a result, the evaluation of approximate computing techniques on scientific applications has been inadequate. Furthermore, there are limited studies on the crucial interaction between approximate computing techniques and the execution of parallel scientific applications. Since parallel execution remains an integral part of HPC, it becomes critical to investigate the applicability of approximation techniques on parallel HPC applications.

In this work, we analyze the interaction between state-of-the-art software-based approximate computing techniques and HPC scientific applications. Our work focuses on kernels, proxy, and mini-applications that use OpenMP to express parallelism. We present

This work is licensed under a Creative Commons Attribution International 4.0 License.

SC ’21, November 14–19, 2021, St. Louis, MO, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8442-1/21/11...\$15.00
<https://doi.org/10.1145/3458817.3476216>

High Performance Approximate Computing (HPAC). HPAC provides extensions to the LLVM/Clang compiler and runtime libraries to support approximate computing. The HPAC-extensions are composable with OpenMP parallel applications. This composability with OpenMP allows us to integrate and evaluate AC techniques on widely-used OpenMP HPC benchmarks. Our approach incorporates pragma-based annotations, code transformations and, analysis to enable developers to identify approximation opportunities in their applications. HPAC implements approximate computing techniques and analysis tools to find code regions suitable for approximation given performance and accuracy criteria

We use HPAC to study a crucial unexplored aspect of approximate computing workloads: *the combined influence of parallelism and approximation on the accuracy and performance of codes*; further, we perform our investigation in eight different HPC applications. Our study draws several insights about the effectiveness of approximations on OpenMP HPC benchmarks. Specifically, our main contributions are the following:

- (1) *HPAC*, a new, pragma-based approximate computing model that includes state-of-the-art approximate computing techniques (loop perforation, input/output memoization) and is composable with OpenMP; HPAC enables easy integration with existing OpenMP codes to identify approximation opportunities. It is also extensible to support new approximation techniques, providing a convenient and versatile framework to explore various approximation strategies.
- (2) An implementation of the approximate computing toolchain in the Clang/LLVM compiler, a parallelism-aware supporting runtime, and tooling that helps to identify approximation opportunities, freely available to the community¹. The implementation is based on a production-level compiler and is robust and usable for a wide range of HPC applications, tested and demonstrated with eight HPC programs in our evaluation.
- (3) A thorough analysis of the applicability of approximate computing methods in HPC. The analysis evaluates three commonly used approximate techniques on OpenMP application. The analysis investigates several approximation techniques and the parameters that influence their execution; the results derive new insight and conclusions on the accuracy and performance trade-offs when composing approximations with parallelism.

Our evaluation outlines many helpful findings for programmers to realize the interaction between parallel execution, accuracy, and performance offered by different approximate computing techniques. Our exploration discovers that approximation provides substantial performance gains by reducing the number of memory accesses in the LULESH proxy application. Approximation in the Leukocyte benchmark induces load imbalance during parallel execution, which shows the complex interplay between approximations, performance, and accuracy.

2 BACKGROUND

There are three commonly used state-of-the-art software-based approximate computing techniques: *loop perforation* [29, 30], *input memoization* (iACT [31]), and *output memoization* (TAF [45]),

which we evaluate for applicability and performance on scientific applications.

Loop Perforation is an approximate technique that skips loop iterations. We consider the following perforation types: i) *small*: skip one iteration for every n iterations, ii) *large*: execute one iteration for every n iterations, iii) *random*: randomly execute $n\%$ of the total number of iterations, iv) *ini*: skip the first $n\%$ of the total number of iterations, v) *fini*: skip the last $n\%$ of the total number of iterations. Perforation types are configurable based on the skip factor parameter n , where n is an integer for the *small* and *large* types and a real number in the interval $[0.0, 100.0]$ for the rest.

Input approximate memoization (iACT) is derived from classic memoization techniques. These techniques cache results of time-consuming function calls to return them when the same input occurs again. iACT relaxes strict equality on inputs and considers two inputs to be the same if their euclidean distance is within a user-defined value *Dist*. The implementation maintains a table of size *tSize* for every call-site and populates it with input values and their corresponding outputs. If a new input is within a distance *Dist* to an existing input value in the memoization table, iACT skips execution of the function and returns the cached output value. If the input does not satisfy the distance criterion, then iACT executes the function and stores the input and the computed output in the table.

Temporal Approximation Function Memoization (TAF) is based on the assumption that functions exhibit temporal output locality, where consecutive invocations of the same function tend to produce similar results. TAF uses the relative standard deviation of output values in a sliding window $\vec{h}$ of size *hSize* as the metric for quantifying temporal locality. Relative standard deviation (RSD) is given by $RSD(\vec{h}) = \mu_{\vec{h}} / \sigma_{\vec{h}}$, where $\mu_{\vec{h}}$ is the mean and $\sigma_{\vec{h}}$ is the standard deviation for the output values in $\vec{h}$. When the $RSD(\vec{h})$ is within a configurable steadiness threshold *stable*, the function is deemed to be in a temporal locality execution phase. Given the temporal locality criterion is satisfied, the subsequent *pSize* function invocations, where *pSize* is the configurable size of a so-called prediction window, skip executing the function and instead return the last accurately computed output. Following those *pSize* approximating function invocations, TAF falls back to the sliding window of accurate execution to re-assess temporal locality.

An application can possibly utilize different approximate techniques for different code regions, and each of those techniques consists of several configuration parameters, such as the skip factor for loop perforation, the distance threshold for input memoization, and the temporal locality window size or prediction window size for output memoization. Selecting the parameter values for the combination of approximate techniques and code regions, while satisfying expected accuracy and performance criteria, is very challenging and practically infeasible to be done manually. Our HPAC framework provides automated tooling to conduct accuracy vs. performance trade-off evaluations on this large search space, in order to identify approximate configurations that maximize speedup subject to accuracy conditions.

¹ <https://github.com/LLNL/HPAC>

Figure 1: HPAC is comprised of two building blocks, the core, and the harness. The core implements the approximate programming model. It extends the Clang/LLVM compiler and provides runtime support. The harness facilitates easy exploration of the approximate design space.

```
1 #pragma approx [perfo(type : expr)] [memo(type: expr-  
list)] [out(locator-list)] [in(locator-list)] [  
inout(locator-list)]
```

Listing 1: Syntax of HPAC pragma.

3 HPAC DESIGN AND IMPLEMENTATION

Figure 1 shows the overall design of HPAC. HPAC specifies distinct components for realizing and evaluating approximate computing techniques on HPC applications. It defines: (1) an annotation-based programming model that operates both standalone and in composition with OpenMP parallelism, (2) the HPAC compiler extensions, that implement compiler support for the HPAC approximate programming model, (3) the HPAC runtime, which implements the runtime support for the execution of approximation-enabled applications, (4) the HPAC Pre-processor script, which creates the approximation design space, and (5) the HPAC Driver script, which automates the accuracy and performance exploration and analysis of approximation techniques on HPC applications. The compiler extensions and the runtime system of HPAC correspond to the core component of HPAC. The developer can use the HPAC core to compile and execute an HPAC-annotated application independently from the remaining utilities. The HPAC preprocessor and driver scripts compose the harness component of HPAC. The harness provides the necessary tooling to explore the design space of the performance, accuracy trade-offs of different approximation techniques applied on various code regions.

3.1 HPAC Programming Model

The approximation programming model of HPAC is based on pragma annotations that apply to code regions of the annotated program to implement approximation semantics. Similar to other annotation-based programming models, such as OpenMP and OpenACC, the

HPAC Programming Model is minimally disruptive to the development process as it does not require any other code transformations besides the addition of the pragma annotation.

Listing 1 presents the syntax of the pragma annotation. The approx directive specifies an approximated region and subsequent clauses define applicable approximation techniques for that region, its parameters, and helper data. Specifically, the clause *perfo* defines loop perforation, applicable to loop regions, and the *memo* clause defines approximating memoization, which is applicable to the next statement in the code, be it a function call, a compound statement in braces, or any other statement including loops.

The *perfo* clause expects two mandatory fields separated with a colon. The first field, a keyword, specifies the perforation type: *large*, *small*, *rand*, *ini*, *fini*. The second field specifies the skip factor *n* for the perforation type, as defined in Section 2.

The *memo* clause requires a keyword specifying the type of memoization. The keyword *iact* specifies input memoization as in *iACT*, and the keyword *taf* specifies output memoization as in *TAF*. The memoization type keyword is followed by a colon-separated list of expressions that correspond to parameters for this type. For *iACT*, the first expression defines the table size (*tSize*), and the second defines the minimum distance threshold *Dist* for input similarity (Section 2). For *TAF*, the expression list defines the sliding window size (*hSize*), the steadiness threshold (*stable*), and the prediction window size (*pSize*) (Section 2). Memoization also requires *in* and *out* modifiers to specify the input and output data of the approximated region. Those modifiers take as arguments a comma-separated list of program variables, either scalar or vectors, or sections of vectors. For example, a user can specify *in*(*X*[*offset*:*num*]), which defines that *num* memory locations of the variable *X* from offset *offset* are inputs to the code region. The syntax for the *out* modifiers is similar. Note that the *out* modifier is required for both memoization types, whereas *in* is used only for *iACT*.

The proposed programming model subsumes previous programming models [20, 31, 40] for loop perforation and memoization under a unified annotation-based model. It is more flexible, especially regarding memoization. The previous memoization approaches only support explicit function calls, whereas HPAC can memoize any kind of code. Furthermore, it supports applying multiple approximation techniques on the same region, decidable at runtime, and nesting approximate directives. For example, it is possible to annotate a loop region with an *approx* directive that specifies all three approximation types (*perforation*, *iACT*, *TAF*) as applicable to the region. Crucially, the programming model is composable with parallel programming models. We chose to support OpenMP in our implementation due to its wide use.

Example: Listing 2 shows an example of composing the HPAC programming model with OpenMP. The example accumulates the values of the 2nd dimension of array *vec* to store them in the array *result*. The *pragma approx* directive in line 3 specifies loop perforation of type *large* with stride 2 for the subsequent loop. This means that the *for* loop on line 5 will execute only even iterations (0, 2, 4 and so on). Further, the OpenMP directive in line 4 specifies this as a parallel, worksharing loop and it is composable with the loop perforation directive preceding it. So, each worker thread in OpenMP will execute the perforated subset of the initial iteration


```

1 void sum_X(float vec[DIM_Y][DIM_X], float result[DIM_Y])
2 {
3     #pragma approx perfo(large:2)
4     #pragma omp parallel for
5     for (int i = 0; i < DIM_Y; i++)
6     #pragma approx memo(in: 100 : 0.5f) \
7         out(result[i]) in(vec[i][0:DIM_X])
8         result[i] = sum(vec[i]);
9 }

```

Listing 2: Example of composing OpenMP with HPAC.

space. Next, the `approx` directive for memoization in line 6 specifies that the loop body statement at line 8 will be approximated using input memoization (iACT) with a table of size $tSize = 100$ and distance threshold $Dist = 0.5f$. The input to this approximated region is the array section `vec[i][0:DIM_X]` and the output the scalar value `result[i]`. From the composition of OpenMP execution with approximation, each OpenMP worker thread will execute the loop body approximated following the input memoization approximation of iACT, skipping calling the `sum` function and instead storing memoized values in the `result[i]` output scalar variable.

This example uses OpenMP with nested approximation and highlights the composability, expressiveness, and flexibility of the HPAC programming model. Notably, previous approaches in the literature lack those qualities that are important for HPC applications. Next, we provide details on the design and implementation of the HPAC compiler that enables the programming model.

3.2 HPAC Compiler Extensions

The HPAC compiler extends the Clang/LLVM compiler to implement the HPAC approximation programming model. In more details, we extend Clang's parser to recognize the pragma-based approximation directive, based on its syntax, and semantic analysis to create a new Abstract Syntax Tree (AST) node for the approximation directive that includes the information on the type of approximation, clauses, modifier, and their related data, as well as the associated code region. Note that an annotated code region is captured as a closure in Clang, so that its accurate, non-approximated version is directly callable as a function. Code generation emits LLVM IR traversing the AST of the program. For approximation, the extended code generation in Clang emits a call to the HPAC runtime entry point function `__approx_exec_call` in place of the approximated region, whose signature is shown in Listing 3. The first argument `accuFn` is the function pointer of the accurate closure function, the argument `perfoFn` stores the function pointer of the perforated loop closure function in case of loop perforation or `null` otherwise, `args` is an opaque pointer for the aggregate storing the arguments to the closure functions (the same either for accurate or perforated closure functions), and the rest of the arguments `memo_type`, `in`, `no_inputs`, `out`, `no_outputs`, `sizes`, `threshold` relate to memoization approximation. We elaborate on code generation next.

Code generation for loop perforation. Our implementation supports perforation for *canonical loops*, with the same definition of canonical form as in the OpenMP specification². Code generation for loop perforation is a two-step process. Semantic analysis for the

```

1 typedef struct {
2     void *ptr;
3     size_t num_elem;
4     size_t sz_elem;
5     int8_t data_type;
6 } approx_var_info_t ;
7
8 typedef enum {
9     MEMO_IACT,
10    MEMO_TAF
11 } memo_t;
12
13 void __approx_exec_call(
14     void (*accuFn)(void *),
15     void (*perfoFn)(void *),
16     void *args, memo_t memo_type,
17     approx_var_info_t *in, int no_inputs,
18     approx_var_info_t *out, int no_outputs,
19     int sizes[2], float threshold);

```

Listing 3: Data structures and the runtime API function call which serves as the main entry point of the approximate runtime system.

loop perforation directive verifies that the associated loop region is in canonical form in order to extract loop control information which includes the loop counter, lower and upper iteration bounds, the loop conditional, the increment step, and the loop body. Following, semantic analysis simplifies loop control by generating an equivalent loop control tuple, where the iteration variable is initialized to 0, the increment step is 1, and the loop conditional is derived from the original lower, upper iteration bounds and increment step. This simplified form helps to perform perforation transformations in code generation and composing perforated loops with OpenMP.

Code generation applies perforating transformation on the simplified loop control variables, depending on the chosen perforation type (`large`, `small`, `rand`, `ini`, `fini`), to emit the perforated loop as an alternative to the accurate closure function to pass to the runtime entry point function. For `rand` perforation, code generation requires runtime support via a helper runtime function prototyped as `bool __approx_skip_iteration(float pr)`. This function takes the random skip percentage as an argument and returns `true` or `false`, for skipping or executing an iteration. Code generation emits a conditional statement before the execution of the loop body at every iteration that calls this helper function and skips loop body execution when the call evaluates to `true` or otherwise falls through to execute it as normal.

Our current implementation supports composing a loop perforation directive with OpenMP parallel worksharing-loop directives, either nested under a parallel region, as in `#pragma omp for`, or combined in a parallel region directive, as in `#pragma omp parallel for`. Semantic analysis for loop perforation identifies when the associated loop is an OpenMP worksharing-loop and performs canonical analysis and simplification on its associated loop region. Following, code generation constructs the perforated loop as in standalone mode, encapsulates it in a new OpenMP worksharing loop construct that replicates the definition of the identified one, and emits a closure containing it to pass to the runtime.

²<https://www.openmp.org/spec-html/5.0/openmpsu40.html#x63-1260002.9.1>

Code generation for memoization. Differently from loop perforation where the approximation is implemented by code generation transformations, memoization is implemented in the runtime. Thus, code generation for memoization mainly passes over information on the input and output data to the runtime. Specifically, code generation passes the memoization type specified in a memo clause as one of the constants of type `memo_t`, defined in Listing 3. Moreover, it stores the parameters of `in`, `out` clauses in arrays of descriptor structs of type `approx_var_info_t`, shown in Listing 3. Each struct entry stores a pointer to the parameter’s memory location, the number of elements, its size and the data type of each element. Code generation allocates those struct arrays on the stack and fills them by using the expressions specified in the `in`, `out` clauses of the `approx` directive, observing any offsets and sizing definitions for array sections. Additionally, it emits parameter values or the sliding and prediction window sizes of the TAF memoization type, or the memoization table size `tSize` for iACT, `in` sizes. Lastly, it emits the memoization threshold value in the `threshold` argument of the runtime entry point, which corresponds to distance `Dist` for iACT and stability threshold `stable` for TAF. Based on those values, the HPAC runtime performs the memoization approximation. We elaborate on the runtime in the next section.

3.3 HPAC Runtime System

Given the interface to the runtime has been presented in the previous section, we will focus the discussion on the internal operation of the runtime system.

Loop perforation. For loop perforation, the runtime simply calls the closure function pointer `perfoFn`, which encapsulates the perforated loop. No other involvement from the runtime is necessary since the compiler’s code generation has implemented the perforating transformation on the loop.

Input memoization iACT. The implementation of the runtime for input memoization uses 2 associated arrays, `InputArray` and `OutputArray` for every approximated code region. The array `InputArray` stores input values for the approximation closure function while the array `OutputArray` stores corresponding output values. On entry through the `__approx_exec_call`, the runtime sweeps the `InputArray` for the code region to compute the minimum euclidean distance between the input values of the current invocation and input values in the input array that were encountered at previous invocations of the same code region. If the minimum distance is less than the user-provided threshold `Dist`, the runtime skips the execution of the original code region, thus does not call `accFn`, but instead copies out the output values from the corresponding `OutputArray` entry. If there is no entry in the `InputArray` to satisfy the distance criterion, the runtime invokes the `accFn` function pointer to execute the region accurately and fills the `InputArray` with the region’s input and the `OutputArray` with the resulting output, given there is free space.

Output memoization TAF. The runtime implements TAF as a state machine with two possible states: `COMPUTE` and `SKIP`. It maintains a history array per code region that contains the previous `hSize` number of accurately computed output values. At initialization, the history array is empty and by default TAF is on the `COMPUTE`. While TAF is in the `COMPUTE` state, it executes the accurate

closure function pointer `accFn` for the region and stores its output in history. Then, it calculates the relative standard deviation (*RSD*) for the output data stored in history and if *RSD* is less than the user-provided *stable* threshold, TAF transitions to the `SKIP` state. When in the `SKIP` state, TAF skips the execution of the region and instead fills its output with the most recent output value in history. TAF stays in the `SKIP` state for a user-provided *pSize* number of consecutive invocations of the approximated region before it falls back to `COMPUTE`.

To make the runtime parallelism-aware and compatible with OpenMP execution, its implementation stores any global data structures in thread local storage. Thus, each parallel thread contains private iACT memoization arrays, a private TAF-state machine, and associated data structures. Although this implementation choice increases memory consumption, it avoids any synchronization among threads during execution and contributes to higher scalability, as we will discuss in the evaluation section.

3.4 HPAC Exploration and Analysis Tools

HPAC also provides tools for the exploration and analysis of the trade-offs between accuracy and performance of applications. In more detail, HPAC includes a harness that allows users to annotate multiple code regions once and automatically test performance and quality trade-offs for them for a set of configurations. HPAC applies different approximation techniques to each annotated code region. For each pair of approximation techniques and code region, HPAC tests distinct parameter values. The harness expects the application to write the final Quantity of Interest (QoI) to a file. HPAC computes the accuracy loss by comparing the QoI of an accurate execution with an approximate one.

<pre>#pragma approx in(x,y)\ out(z) perfo memo for(i = 0; i < N; i++) z +=x[i]*y[i];</pre>	<pre>for(i = 0; i < N; i++) z +=x[i]*y[i];</pre>
(a) Original Code	(b) Accurate code instance
<pre>#pragma approx\ in(x,y) out(z) \ memo(__PARAMS__) for(i = 0; i < N; i++) z +=x[i]*y[i];</pre>	<pre>#pragma approx \ perfo(__PARAMS__) for(i = 0; i < N; i++) z +=x[i]*y[i];</pre>
(c) Memoization code instance	(d) Perforation code instance

Listing 4: The user provides to the HPAC-harness an application annotated with the abstract form, presented in Listing 4a. In this form, the developer does not provide any parameters to the approximation techniques. During the pre-processing step, the HPAC-preprocessor script creates a separate code instance for each approximate code region and each applicable approximate technique. Noticeably, it creates an additional code instance without any HPAC annotations (Listing 4b). This instance is the baseline for performance and accuracy comparisons. The HPAC-driver defines the `__PARAMS__` macro-variable (Listings 4c, 4d) appropriately to select different approximation parameters. Then, it invokes the extended CLANG/LLVM compiler.

The implementation of the harness includes two separate software components: the preprocessor and the driver (Figure 1). The preprocessor expects the application source code annotated with approximation directives that define multiple, applicable approximations for the same code region in abstract form. Listing 4a presents an example of the abstract form. In the example code, the perforation and memoization parameters are blank. However, the in/out clauses are specified since they are part of the region specification. The preprocessor parses the source code files and fills the parameters of approximation clauses with macro-variables. The harness driver defines the values of the macro-variables and then compiles the code. For each approximation technique, the harness creates multiple, different instances of the source code. Each source code instance applies a distinct approximation technique for a specific code region. So, in the example, there are 2 approximation definitions, memo and perfo. Thus, the harness will create 3 separate source code instances, presented in Listing 4, one for each approximation technique plus an extra one without any approximate annotations. The accurate code instance is the baseline to perform comparisons of performance and output quality. After the preprocessor finishes generating those instances, it passes their description to the driver. Listing 4 presents the intermediate output of the preprocessor.

The HPAC driver defines approximation parameters for each technique before building each instance. Due to this separation of parameter definitions and compilation, the driver can apply different approximation techniques on the annotated code regions and define distinct parameters for each one. Also, the driver accepts an input configuration file containing the parameter values to apply for each approximate technique. Following building, the driver executes all binaries and computes the approximation error by comparing the results of the accurate instance and the approximated ones. The error, the approximation technique used, the code region, and the performance of each execution are stored in a pandas [24] dataframe. The user can access the dataframe with standard Python libraries to view the results.

4 EVALUATION

In this section, we describe our analysis results.

4.1 Methodology

To compare the accuracy loss we use two error metrics, the Mean Absolute Percentage Error (MAPE), and the Miss-Classification-Rate (MCR) defined in Equations 1 and Equation 2, respectively. $\vec{Acc}$ and $\vec{Approx}$ are two vectors of size N containing quantities of interest for each application. The operator $\mathcal{I}$ returns 1 when the condition is true.

$$MAPE(\vec{Acc}, \vec{Approx}) = \frac{1}{N} \sum_{i=1}^N \frac{|\text{Acc}_i - \text{Approx}_i|}{\text{Acc}_i} \quad (1)$$

$$MCR(\vec{Acc}, \vec{Approx}) = \frac{1}{N} \sum_{i=1}^N \mathcal{I}(\text{Acc}_i \neq \text{Approx}_i) \quad (2)$$

Table 1 lists the benchmarks used in our evaluation, the error metric, and the QoI for each application. The QoI differs among the applications. Comparing error values is meaningful only for

Table 1: List of applications evaluated in this work.

Benchmark	Application Description	Quantity of Interest (QoI)
LULESH [21]	An proxy that approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh.	The final origin energy.
HPCCG [19]	A PDE application and preconditioned conjugate gradient solver that solves a linear system.	The final residual of the solver.
Leukocyte [9]	The application detects and tracks rolling leukocytes (white blood cells) in video microscopy of blood vessels.	The final location of the tracked leukocytes.
LavaMD [9]	The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space.	The location and the force of the different particles in the simulation.
CFD [9]	An unstructured grid finite volume solver for the three-dimensional Euler equations applied to compressible flows.	The location, the momentum and the density of the simulated particles.
K-Means [9]	K-Means is a clustering algorithm and assigns observations to their closest cluster.	The cluster of each observation.
Particle-Filter [9]	The application is a statistical estimation of the location of a target object given noisy measurements.	The location of the target elements.
Blackscholes [5]	This application calculates the price of a portfolio of European stock options analytically.	The computed prices.

the same application, not across applications. All the applications use MAPE as the error metric, except K-Means, which uses MCR. In all applications, we store the quantity of interest in a file. After the end of each experiment, the HPAC-driver applies the error metric between the approximate file output and an accurate one to compute the error.

We profile all the benchmarks to identify their hotspots, i.e., regions contributing significantly to the application’s overall execution time. We use this profiling information to identify code regions that are amenable to approximations within each hotspot. We apply the approximate abstract pragma annotations — presented in Section 3.4 — on the whole hotspot regions or in regions nested in the hotspots. HPAC does not require the profiling step. However, the step limits the analysis search space only on code regions that present large performance overheads.

We provide the source code files of each application to the HPAC-harness. The HPAC-preprocessor out of the abstract pragma annotations creates a separate code instance for each pair of code region-applicable approximate technique as discussed in Section 3.4. Each code instance approximates a code region with an approximation technique. We create an additional code instance, the accurate one, which we use as a baseline.

The HPAC-preprocessor forwards to the HPAC-driver the code instances description. In turn, the HPAC-driver defines different approximation parameters via the macro-variables — discussed in Section 3.4 — and compiles the code instances into separate binaries. Each binary corresponds to a unique combination of: (i) approximate code region, (ii) approximation technique, and (iii) approximation technique parameters. In the rest of the document we refer to these binaries as configurations. Within the configurations there

Table 2: Approximation parameters used for our evaluation.

Perforation			Approximate-Memoization		
Type	Parameter	Tested Values	Type	Parameter	Tested Values
large	<i>step</i>	2, 4, 8, 16	iACT	<i>tSize</i>	5, 10, ..., 30
small	<i>step</i>	2, 4, 8, 16		<i>Dist</i>	10, 20, ..., 90
rand	<i>ratio</i>	10, 20, ..., 90	TAF	<i>hSize</i>	2, 3, 4, 5
ini	<i>ratio</i>	10, 20, ..., 90		<i>pSize</i>	2, 3, 4, 5, 6, 7, 8
fini	<i>ratio</i>	10, 20, ..., 90		<i>stable</i>	0.3, 0.6, ..., 3.0

Figure 2: Blackscholes: speedup vs quality loss. Different markers indicate the approximation technique, and the colors indicate the approximated code region. Performance gains are observed only when using perforation on the BlsEq code region. The BlsEq region corresponds to the whole computation of the Blackscholes formula

is also an accurate configuration used as a baseline. We experiment with different values of parameters, listed in Table 2. In our evaluation, HPCCG, LULESH, and Particlefilter need to explore a broader range of parameter values for iACT and TAF to meet their activation criteria. Therefore, for these applications we increase the range of values for the *Dist* and *stable* parameters for iACT and TAF, respectively.

We execute the configurations with 1, 16, and 32 threads. We define an experiment as the combination of a configuration and a specific number of threads. Except for random perforation, other experiments always produce the same output and, thus, the same error value. Random perforation induces a non-deterministic behavior. However, in our experiments, we seed the random generator with the same seed value. Therefore, the random perforation experiments for the same configuration result in identical accuracy.

To accommodate for performance variations induced by environmental/system conditions we execute each experiment 8 times. To compute the average execution time of an experiment we select the inner-quantile range of the execution time of the performed experiments. Finally, unless stated otherwise, to compute the speed up of the experiments we compare the average execution time of the accurate experiment versus the approximate experiment for the same number of threads. All the experiments were conducted on an Intel Xeon E5-2695 v4 processor with 128GB of DRAM memory.

4.2 Evaluating Performance Gains and Quality Loss on Code Regions

In this section, we study the performance benefits and accuracy losses of approximate computing techniques. We apply these techniques to different code regions and benchmarks. The plots present the entire domain space, which includes parallel and sequential execution. However, in this section, we focus only on the interaction between approximate code region and approximation technique and we do not elaborate on the effects of parallelism, which is considered separately in Section 4.3.

Blackscholes is a financial analysis application from the PARSEC benchmark suite. The original input file of Blackscholes contains 10M input values. However, the input file contains only 821 unique input values. In such an input pattern, the input memoization technique would always memoize without inducing an error. Therefore, we use a new input file for BlackScholes where it computes the BlackScholes formula on 10M distinct input values.

Figure 2 shows the accuracy vs. performance tradeoff space explored by the HPAC harness. Note that the plot shows the combined data points from sequential and parallel executions. We approximate the following functions: BlsEq, Cndf1, and Cndf2, and exp function invocation. Perforation is only applicable to the *for* loop enclosing the BlsEq function invocation. Hence, it will skip a corresponding BlsEq invocation, resulting in skipping the computation of a stock price.

BlsEq is the most compute-intensive part of the application, and Figure 2 shows that BlsEq is the only function that benefits from any of the tested approximate techniques. iACT does not result in better approximations nor provides any performance gains. This is primarily because BlsEq has multiple scalar inputs with different impacts on the output. However, the distance function considers all the input arguments equally important and does not factor in the impact while computing the distance metric. Surprisingly, perforation results in better quality than TAF while providing similar speedup. Since Blackscholes is a streaming application, it does not exhibit temporal locality, which explains poor accuracy for TAF.

CFD: We annotate the *for* loop within the *compute_flux* function for applying approximation. The loop iterates over the elements of the simulated fluid and computes their flux. Figure 3 shows the accuracy vs. performance trade-off space and the results suggest that CFD is a good candidate for approximations. The results indicate that approximation techniques, specifically perforation and TAF, are able to provide speedup while incurring less than 3% error. On further investigation of the effectiveness of different perforation variants, we observed that the *rand* variant provides the most speedup with the least amount of error. Figure 4 shows the accuracy and performance for the different variants of the perforation approximate technique. The stochasticity introduced by the *rand* variant helps achieve lower error, whereas the rest of the variants skip the same set of elements for every execution of the loop resulting in the slow progress of the simulation and higher error.

HPCCG: The annotated regions for approximation include the inner *for* loop within *sparseMV* kernel, the dot products (DOT, R2), and the *waxpby* kernel. Figure 5 shows the accuracy performance

Figure 3: CFD: speedup vs quality loss. The most effective approximation technique is perforation. Typically, random perforation results to better quality for this application.

Figure 4: Comparison of different variants of perforation for CFD. Random perforation achieves the highest accuracy, but is not as performant as the other perforation variants.

Figure 5: HPCCG: speedup vs quality loss. HPCCG presents is not a good candidate for the tested approximate techniques.

Figure 6: K-Means: speedup vs. quality loss. The results indicate that the FNP code region is a well suited region for approximations as it presents small error values with large performance gains.

Figure 7: Leukocyte: speedup vs. quality loss. Leukocyte, is extremely resilient to approximations as it presents low errors.

tradeoff space for HPCCG. Note that the y-axis is log scale, and the results indicate that HPCCG is very sensitive to approximation.

K-Means: We add approximation pragmas to the functions ED (Euclidean Distance) and FNP (Find Nearest Point). ED computes the euclidean distance between an observation vector and a cluster, and FNP assigns an observation to a cluster. Figure 6 shows the performance vs. accuracy space for these two code regions. Although ED is within the FNP region, results indicate that approximating the overall FNP provides better speedup with lower error. This is due to the fact that approximating the euclidean distance (ED) is more error-prone than approximating the nearest cluster (FNP). Moreover, even a small error in the distance calculation can result in a wrong cluster assignment, thereby affecting the overall output quality.

Leukocyte: This application detects and tracks rolling leukocytes (white blood cells) in a video microscopy of blood vessels. The application iterates over the frames and updates the positions of the white blood cells. The candidate regions for approximation are the following: EE, UCL, FM, FL, and FD. All the regions are approximated using all the three techniques, except for EE, where

Figure 8: Lulesh: speedup vs. quality loss. Approximation in Lulesh can result to speed ups of up to $1.7\times$ in comparison with the accurate execution.

Figure 9: Particlefilter: speedup vs. quality loss. FD finds the index of each particle using a search algorithm. Applying perforation on FD results to significant performance gains with low accuracy loss.

perforation is not applied due to the absence of a for loop within that region. In Figure 7 we present the accuracy vs. performance tradeoff for Leukocyte. This application is extremely resilient to approximations as it achieves significant speedups with very low error. Applying approximate techniques to the main computation function, UCL, provides the most gains, specifically from TAF and perforation approximations. TAF achieves speedup of up to $3.9\times$ with an error of 1.15%, whereas perforation results to speedup of $11\times$ with an error of 1.4%. Note that the parameters of iACT can significantly affect the effectiveness of approximation. Some parameter choices can result in the execution of the UCL region only once at the beginning of the application, which invariably provides great speedup at the cost of introducing a large error.

LULESH: We identified 10 regions where at least one approximation technique is applicable. However, most of them gave only marginal performance gains with approximation. CalcElemVolumeDerivative (VO) and CalcElemFBHourglassForce (HG) showed most significant performance gains, where TAF introduced only 0.1% error and provided speedups of $1.4\times$ and $1.7\times$, respectively.

Figure 10: Speedups for two code regions approximated with TAF for different number of threads. Each violin contains the speed up for the tested parameters in comparison with a pure OpenMP execution with the same number of threads

Particlefilter: Figure 9 shows the performance vs. accuracy loss for Particlefilter. We approximate FD, LH, and WS code regions in this benchmark. Our results show that FD, which performs a linear search to find the index of each particle in a given cumulative density function, has the largest performance gains within a reasonable error. We find that memoization is not a suitable technique for FD, but perforation can give substantial speedup. Particlefilter is a statistical application, hence perforation techniques, especially of the rand variant, tend to provide maximum performance gains without introducing much error.

4.3 Evaluating the Composition of Approximation and Parallelism

In this section, we analyze the combined effect of parallelism and approximation on the accuracy and performance of HPC benchmarks. We discuss results pertaining to LULESH, Leukocyte, and K-Means, which provide the most insight.

LULESH: When approximating the VO or the HG functions in LULESH, we achieve further speedup by increasing the number of OpenMP threads. Figure 10 shows the overall speedup achieved by approximate techniques for different number of threads. We compare the execution time of a pure OpenMP version with an executions of OpenMP + HPAC for the same number of threads. Note that here we show only the data points that are within an error threshold of 0.1%. We attribute this behavior to a reduction in memory accesses due to approximations. In Figure 11 we show the number of load operations performed on the last-level cache with approximation (shown as VO and HG) and without (shown as Acc.) for various thread counts. As we increase the number of threads, the number of cache accesses grows significantly. However, for the approximate versions, this increase in cache accesses with thread counts grows slower and in the case of 32 threads it reduces the number of cache accesses when compared to 16 threads. We observe similar behavior for store operations as well.

K-Means: In Figure 12 we show the speedup offered by the different approximate techniques when applied to the FNP region of K-Means and the speedup of. The Figure presents the speed up of the approximation techniques and of pure OpenMP version against

Figure 11: Comparison of number of load operations on the outer level cache for LULESH. In Lulesh, approximations reduces the number of loads on the outer level cache.

Figure 12: Swarmplots comparing the speedup relative to pure OpenMP with 1 thread for pure OpenMP and OpenMP + HPAC versions for different approximate techniques across different number of threads (1, 16, and 32) for K-Means. We plot only configurations that introduce an error (MAPE) of less than 20%. The performance gains offered by iACT increases as we increase the number of threads. This is because each thread owns and stores the memoization data in their iACT table, which results in an overall increase in the size of the iACT table available for the application

a sequential execution. Note that we include only the data points that introduce an error of less than 20%. The performance gains offered by iACT increases as we increase the number of threads. This is because each thread owns and stores the memoization data in their iACT table, which results in an overall increase in the size of the iACT table available for the application. K-Means can take advantage of larger iACT memoization table enabling for more approximations, which results in speedup and better quality. Table 3 shows that the fraction of approximated executions increases from 17% to 88% as the thread counts increase from 1 to 32, thereby validating our hypothesis. By contrast, TAF did not see the same kind of performance gains when increasing the number of threads. This is primarily because TAF is unable to utilize the increase in the number of threads to improve the approximated execution fraction. In fact, this fraction stays at 75% irrespective of number of threads.

Table 3: Effectiveness of approximation for K-Means. It shows the approximated fraction of executions using iACT, the obtained speed up and the respective quality loss for different number of threads and different table sizes ($tSize$).

$tSize$	Threads:	Fraction (%)			Speed Up ($\times$)			% MCR		
		1	16	32	1	16	32	1	16	32
5		17	86	88	1.9	1.5	2.1	29	9	10
10		40	92	94	1.06	1.3	3.3	40	9	11
15		98	96	98	0.8	2.9	4.9	29	10	11

Figure 13: Swarmplots comparing the speedup relative to pure OpenMP with 1 thread for pure OpenMP and OpenMP + HPAC versions for different approximate techniques across different number of threads (1, 16, and 32) for Leukocyte. All versions use the static scheduling policy. All points above the black line present HPAC configurations which present additional speeds ups. We plot only configurations that introduce an error (MAPE) of less than 5%.

While this justifies why TAF does not have similar performance gains as iACT, the reason for its diminishing gains is explainable by Amdahl's law, which states that the speedup is limited by the part of the execution that cannot benefit from the improvement.

Leukocyte: Figure 13 shows the speedup offered by the approximate techniques when applied to the UCL region of Leukocyte. The Figure presents the speed up of the approximation techniques and of pure OpenMP version against a sequential execution. Note that we include only the data points that introduce an error of less than 5%. Besides the speed-up provided by increasing the number of threads HPAC provides additional speed-ups which can reach up to 25 $\times$. However, the additional speed ups of HPAC decrease as we increase the number of OpenMP threads.

Leukocyte presents poor load balancing. Approximate computing techniques further exacerbated this issue. Figure 14 validates our hypothesis as it shows that the parallelism introduces load imbalance, and approximation further worsens it. Specifically, the HPAC is still faster than the parallel OpenMP version (as presented in Figure 13), however, the magnitude of the speedup is decreased due to the additional load imbalance issues. We experimented with different OpenMP scheduling policies — guided and dynamic —

Figure 14: Comparison of Load Imbalance ($1 - \frac{AVG_w}{MAX_w}$) for accurate and approximate version of Leukocyte for thread counts of 16 and 32 using different OpenMP scheduling policies. Leukocyte suffers from a load imbalance problem, which is exacerbated by applying TAF. Dynamic and guided scheduling policy helps only to a certain extent because of lack of work to distribute. Despite the additional speed-ups introduced by HPAC (Figure 13), the gains of approximation in Leukocyte could be even higher.

using a chunk-size of 1. The dynamic scheduling policy should diminish all load imbalancing issues. However, the video used as input in Leukocyte limits the parallel work to 36 cells. Since in Leukocyte there is not sufficient total work, the OpenMP workers cannot dynamically get jobs. So, the amount of existing work bounds the efficiency of the dynamic scheduling policy.

5 INSIGHTS

In this section we briefly discuss our insights and highlight some of the key points of our analysis.

- (1) Memoization techniques use an activation criterion to decide whether to approximate an invocation or not. This functionality decides the effectiveness of the approximation technique. An ideal activation function can find the balance between frequency of activation, error minimization, and performance gains. Yet it is challenging to implement a generic application independent activation function.
- (2) In iACT the activation criterion is based on the euclidean distance, which is used as a similarity metric. Our analysis shows cases that in many applications this criterion is inadequate. The euclidean distance considers all input variables as equally important and does not factor in the impact of each input on the output. Therefore, it can consider input variables as the same and result in large errors. Moreover, the euclidean distance is time consuming to calculate, especially in the case of code regions that operate on vectors, inflating the total execution time. As future work, we will research dimensionality reduction and feature selection techniques to address those limitations.
- (3) Memoization spends more resources to approximate than perforation. However, the performance-quality trade-offs are not straight forward. Our evaluation shows that perforation techniques, which skip computations without additional overhead, can result in better performance and quality of the output.

- (4) TAF should not be used in applications that do not exhibit any temporal locality, as was the case for Blackscholes which had streaming data pattern.
- (5) In the case of perforation, the largest speedups are observed when using the ini or fini variants. In terms of error though, the rand perforation variant consistently produced the least error.
- (6) Based on our user experience with the approximation techniques, perforation is the most intuitive one. Although for us (the users) it was hard to fully grasp the implications of perforation on the application error, it was easy to understand the potential performance improvement due to perforation. The more one increases the skip factor the more performance one gains. On the other hand, memoization techniques present a complex interplay between their parameters which is not always easy to predict.
- (7) Approximation, in general, can result in significant performance gains ($1.7\times$ LULESH) and further increase performance under parallel execution. Yet, the interaction between parallelism and approximation is application dependent. Nonetheless, our experimentation shows that performance improvements are achievable by composing approximations with parallel execution. There are cases of using OpenMP dynamic scheduling policies to alleviate load balancing issues arising due to approximation techniques.
- (8) The performance gains of approximations follow Amdahl's law, which implies that the speedup due to approximations is limited by the part of the execution that cannot benefit from those approximations.

6 RELATED WORK

Approximation Techniques: Several general-purpose approximate computing techniques have been proposed in the literature. In [25, 26, 36] the authors present a method to skip entire parallel tasks. In [29, 30, 44] the authors present loop perforation, a technique that skips iterations of a loop. In [3, 49], the authors present methods to automatically create alternative approximate functions by randomly mutating the code. In [13, 31, 38, 45] the authors present approximate memoizations. Other approaches use mixed-precision techniques [22, 23, 27, 37] to reduce the execution time of floating-point instructions by using less precise representations. Our work is complementary to those as we enable approximation techniques using a new framework and study the interactions of approximation on parallel HPC applications.

Parallel Approximate Programming Models: The authors in [33, 34] propose classifying OpenMP tasks depending on the instruction types they execute. Depending on this characterization, the tasks are scheduled to be executed on unreliable hardware, which is more energy-efficient. A similar approach in [32] executes parallel tasks on undervolted, and thus, unreliable CPUs to study the potential energy gains. A programming model, a runtime system, and an offline energy modeling approach is presented in [46–48]. In these works, the developer manually specifies alternate approximate versions of tasks and ranks the tasks depending on their contribution to the final output quality. During execution time, the runtime system uses this information and executes approximately

the least significant tasks. In addition, these works tightly couple approximations with parallelism and study the overall gains of approximations for improving energy efficiency. In contrast, we support already existing approximate computing techniques and do not rely on manual approximations. Furthermore, in this work we compose approximation with OpenMP annotations. Semantically, this is an important distinction, as our approach does not rely on parallelism but co-exists with it. The authors in [1, 6, 7] exploit either low-rank approximations or mixed precision computations to improve the performance of the Cholesky algorithm. They describe the extensions performed on PaRSEC [12], a task based runtime system, to accommodate for issues induced by approximation, such as load balancing. Similar to this work, they study the trade-offs between parallelism and accuracy, however they focus on a specific algorithm, whereas in this work we focus on a programming model which can introduce approximation in an abstract and general way. **Approximate Frameworks:** GREEN [4] is an API for loop and function level approximations. It uses perforation on loops and defines alternative approximate functions of an original accurate function. The API includes calibration functions that build application error models. SAGE [39] is a compiler and a runtime system targeting GPUs which allows automatic generation of approximate kernels in machine learning and image processing applications. ACCEPT [40] implements in C/C++ the Enerj [41] type-qualifiers approximation extensions. It provides a quality aware auto-tuner that automatically chooses the best approximation strategies. ApproxHPVM [42] proposes a compiler intermediate representation, which allows accuracy aware compiler optimizations for ML applications, implemented in the Keras programming model. The authors in [43] extend ApproxHPVM with an auto-tuning framework for approximate aware optimizations of tensor-based applications.

The developer uses PetaBricks [2, 3, 11] and provides alternate algorithm implementations with different accuracies. Rely [8] and Chisel [28] provide a function level specification that expresses the maximum probability with which a function can produce erroneous results. Their techniques require extensions on the Instruction Set Architecture and the Memory hierarchy. Each approximate component has a probability to manifest an error. Their approaches map the software specifications to the hardware implementations. In contrast to these works, we study also parallelism and provide a framework in order to help developers to apply approximation techniques to applications in a structured and guided manner.

7 CONCLUSION

The main goal of this work was to understand how approximation composes with OpenMP-level parallelism. We presented an exhaustive evaluation, studying the effectiveness of approximate algorithms on eight different OpenMP HPC benchmarks and characterized their effect on the application accuracy and performance. We developed HPAC, a compiler, and runtime-based framework, which enabled us to conduct experimental studies to understand the trade-offs of accuracy and performance. Our framework allows users to annotate code regions of interest and implements the most commonly used approximate techniques. Also, it is extensible by design and allows for seamless integration of new approximation techniques. With the support of HPAC, developers can effortlessly

evaluate several approximate computing methods for different regions of code and select the approximation technique that meets their performance and accuracy criteria.

In the future, we will study the effect of approximations when applied to different implementations and algorithms of the same applications. Moreover, we plan to implement load imbalance-aware approximation techniques.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. The views and opinions of the authors do not necessarily reflect those of the U.S. government or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein. This work was prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-821216) and was supported by the LLNL-LDRD Program under Project No. 20-ERD-043.

REFERENCES

- [1] Sameh Abdulah, Qinglei Cao, Yu Pei, George Bosilca, Jack Dongarra, Marc G. Anton, David E. Keyes, Hatem Ltaief, and Ying Sun. 2021. *Accelerating Geostatistical Modeling and Prediction With Mixed-Precision Computations: A High-Productivity Approach with PaRSEC*. Technical Report.
- [2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. Petabricks: A language and compiler for algorithmic choice. *ACM SIGPLAN Notices* 44, 6 (2009), 38–49.
- [3] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 85–96.
- [4] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 198–209.
- [5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.
- [6] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David E. Keyes, and Jack Dongarra. 2020. Leveraging PaRSEC Runtime Support to Tackle Challenging 3D Data-Sparse Matrix Problems. (2020).
- [7] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2020. Extreme-scale task-based cholesky factorization toward climate and weather prediction applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–11.
- [8] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. *ACM SIGPLAN Notices* 48, 10 (2013), 33–52.
- [9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. <https://doi.org/10.1109/IISWC.2009.5306797>
- [10] Ron S. Dembo, Stanley C. Eisenstat, and Trond Steihaug. 1982. Inexact newton methods. *SIAM Journal on Numerical analysis* 19, 2 (1982), 400–408.
- [11] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. *ACM SIGPLAN Notices* 50, 6 (2015), 379–390.
- [12] Jack Dongarra, G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, and T. Herault. 2013. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *IEEE Computing in Science and Engineering* 15 (2013), 36.
- [13] Rudolf Eigenmann. 2017. HiPA: history-based piecewise approximation for functions. In *Proceedings of the International Conference on Supercomputing*. 1–10.
- [14] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the*

- seventeenth international conference on Architectural Support for Programming Languages and Operating Systems. 301–312.
- [15] Hadi Esmailzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 449–460.
 - [16] Saman Froehlich, Daniel Große, and Rolf Drechsler. 2018. Towards reversed approximate hardware design. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 665–671.
 - [17] Daniele Funaro. 2008. *Polynomial approximation of differential equations*. Vol. 8. Springer Science & Business Media.
 - [18] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 615–626.
 - [19] Michael A. Heroux. 2017. High Performance Computing Conjugate Gradients: The original Mantevo miniapp. <https://github.com/Mantevo/HPCCG>
 - [20] Henry Hoffmann, Sasa Misailovic, Stelios Sidirolglou, Anant Agarwal, and Martin Rinard. 2009. Using code perforation to improve performance, reduce energy consumption, and respond to failures. (2009).
 - [21] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973, 1–9 pages.
 - [22] Ignacio Laguna, Paul C Wood, Ranvijay Singh, and Saurabh Bagchi. 2019. Gpumixer: Performance-driven floating-point tuning for gpu scientific applications. In *International Conference on High Performance Computing*. Springer, 227–246.
 - [23] Michael O Lam, Tristan Vanderbruggen, Harshitha Menon, and Markus Schordan. 2019. Tool integration for source-level mixed precision. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 27–35.
 - [24] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing* 14, 9 (2011), 1–9.
 - [25] Jiayuan Meng, Srmat Chakradhar, and Anand Raghunathan. 2009. Best-effort parallel execution framework for recognition and mining applications. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–12.
 - [26] Jiayuan Mengte, Anand Raghunathan, Srmat Chakradhar, and Surendra Byna. 2010. Exploiting the forgiving nature of applications for scalable parallel execution. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.
 - [27] Harshitha Menon, Michael O Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: algorithmic differentiation applied to floating-point precision tuning. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 614–626.
 - [28] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. *ACM Sigplan Notices* 49, 10 (2014), 309–328.
 - [29] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. 2011. Probabilistically accurate program transformations. In *International Static Analysis Symposium*. Springer, 316–333.
 - [30] Sasa Misailovic, Stelios Sidirolglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1*. 25–34.
 - [31] Asit K Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*. 52.
 - [32] Konstantinos Parasyris, Vassilis Vassiliadis, Christos D Antonopoulos, Spyros Lalīs, and Nikolaos Bellas. 2017. Significance-aware program execution on unreliable hardware. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.
 - [33] A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, and L. Benini. 2013. Variation-tolerant OpenMP tasking on tightly-coupled processor clusters. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 541–546. <https://doi.org/10.7873/DATE.2013.121>
 - [34] Abbas Rahimi, Andrea Marongiu, Rajesh K Gupta, and Luca Benini. 2013. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.
 - [35] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, Jorg Henkel, and Jörg Henkel. 2016. Architectural-space exploration of approximate multipliers. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.
 - [36] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*. 324–334.
 - [37] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
 - [38] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 35–50.
 - [39] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–24.
 - [40] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* 1, 2 (2015).
 - [41] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. *ACM SIGPLAN Notices* 46, 6 (2011), 164–174.
 - [42] Hashim Sharif, Prakash Srivastava, Muhammad Huzaifa, Maria Kotsifakou, Keyur Joshi, Yasmin Sarita, Nathan Zhao, Vikram S Adve, Sasa Misailovic, and Sarita V Adve. 2019. ApproxHPVM: a portable compiler IR for accuracy-aware optimizations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 186–1.
 - [43] Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Akash Kothari, Ben Schreiber, Elizabeth Wang, Yasmin Sarita, Nathan Zhao, Keyur Joshi, Vikram S Adve, et al. 2021. ApproxTuner: a compiler and runtime system for adaptive approximations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 262–277.
 - [44] Stelios Sidirolglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 124–134.
 - [45] Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni. 2018. Temporal approximate function memoization. *IEEE Micro* 38, 4 (2018), 60–70.
 - [46] Vassilis Vassiliadis, Charalampos Chaliōs, Konstantinos Parasyris, Christos D Antonopoulos, Spyros Lalīs, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2015. A significance-driven programming framework for energy-constrained approximate computing. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. 1–8.
 - [47] Vassilis Vassiliadis, Charalampos Chaliōs, Konstantinos Parasyris, Christos D Antonopoulos, Spyros Lalīs, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2016. Exploiting significance of computations for energy-constrained approximate computing. *International Journal of Parallel Programming* 44, 5 (2016), 1078–1098.
 - [48] Vassilis Vassiliadis, Konstantinos Parasyris, Charalampos Chaliōs, Christos D Antonopoulos, Spyros Lalīs, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2015. A programming model and runtime system for significance-aware energy-efficient computing. *ACM SIGPLAN Notices* 50, 8 (2015), 275–276.
 - [49] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. 2012. Randomized accuracy-aware program transformations for efficient approximate computations. *ACM SIGPLAN Notices* 47, 1 (2012), 441–454.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We extend each application in the paper to use our pragmas. We compile them using our compiler version. Each application is executed 8 times, for each given configuration and we drop the statistical outliers outside of the 25

We run modified versions of the rodihnia, suite including HPCCG <https://github.com/Mantevo/HPCCG>, lulesh <https://github.com/LLNL/LULESH>, blackscsholes <https://parsec.cs.princeton.edu/download.htm>.

We use the following commands/inputs to execute the applications.

- **Lulesh:** Command line parameters: -s 80 -i 10
- **HPCCG:** Command line parameters: 64 64 64
- **Kmeans:** Command line parameters: -i kdd_bin -k 5 -l 30 -n "X". kdd_bin is the input file and X was the number of threads we tested.
- **LavaMD:** Command line parameters: -cores 4 -boxes1d 10
- **blackscholes** we created a new larger input with random numbers. Command line parameters: "X" input X corresponds to the number of threads.
- **leukocyte:** Command line parameters: 20 "X"
../data/leukocyte/testfile.avi
- **particle-filter:** Command Line : -x 128 -y 128 -z 20 -np 10000

Finally, the README file in the GitHub repository <https://github.com/koparasy/HPAC> contains detailed instructions on how to replicate the experimental procedure performed for our evaluation.

Author-Created or Modified Artifacts:

Persistent ID: <https://doi.org/10.5281/zenodo.5167980>

Artifact name: HPAC

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon E5-2695v4

Operating systems and versions: Linux 3.10.0-1160.15.2.1chaos.ch6.x86_64

Compilers and versions: clang version 11.0.0

Applications and versions: Rodihnia 3.1, Parsec 3.0

Libraries and versions: OpenMP 5.0, pandas 1.2.2, numpy 1.20.1, logging 0.5.1.2, yaml 5.4.1

Input datasets and versions: Default input sets and parameters of the benchmark suites we used besides blackscholes. In blackscholes we create a random input data set.