Noname manuscript No.
(will be inserted by the editor)

Exploiting significance of computations and profile-driven regression for energy-constrained approximate computing

Vassilis Vassiliadis · Charalampos Chalios · Konstantinos Parasyris · Christos D. Antonopoulos · Spyros Lalis · Nikolaos Bellas · Hans Vandierendonck · Dimitrios S. Nikolopoulos

Received: date / Accepted: date

Abstract Approximate execution is a viable technique for environments with energy constraints, provided that applications are given the mechanisms to produce outputs of the highest possible quality within the available energy budget.

This paper introduces a framework for energy-constrained execution with controlled and graceful quality loss. A simple programming model allows developers to structure the computation in different tasks, and to express the relative importance of these tasks for the quality of the end result. For non-significant tasks, the developer can also supply less costly, approximate versions. The target energy consumption for a given execution is specified when the application is launched. A significance-aware runtime system employs an application-specific analytical energy model to decide how many cores to use for the execution, the operating frequency for these cores, as well as the degree of task approximation, so as to maximize the quality of the output while meeting the user-specified energy constraints.

Evaluation on a dual-socket 16-core Intel platform using 9 kernels and applications shows that the proposed framework performs very close to an oracle always selecting the optimal configuration, both in terms of energy efficiency and quality of results. Also, a comparison with loop perforation (a well-known compile-time approximation technique), shows that the proposed framework results in significantly higher quality for the same energy budget.

Keywords Approximate computing, significance, energy efficiency, modeling

V. Vassiliadis
University of Thessaly, Greece and
Centre for Research and Technology Hellas,
E-mail: vasiliad@uth.gr

Charalampos Chalios
Queen's University of Belfast, United Kingdom
E-mail: cchalios01@qub.ac.uk

K. Parasyris
University of Thessaly, Greece and
Centre for Research and Technology Hellas,
E-mail: koparasy@uth.gr

C. D. Antonopoulos
University of Thessaly, Greece and
Centre for Research and Technology Hellas,
E-mail: cda@uth.gr

S. Lalis
University of Thessaly, Greece and
Centre for Research and Technology Hellas,
E-mail: lalis@uth.gr

N. Bellas
University of Thessaly, Greece and
Centre for Research and Technology Hellas,
E-mail: nbellas@uth.gr

H. Vandierendonck
Queen's University of Belfast, United Kingdom
E-mail: h.vandierendonck@qub.ac.uk

D. S. Nikolopoulos
Queen's University of Belfast, United Kingdom
E-mail: d.nikolopoulos@qub.ac.uk

1 Introduction

Energy consumption is a fundamental challenge for the entire computing ecosystem, from the tetherless devices that must operate in severely energy-constrained environments to the datacenters that must tame the data deluge. Large-scale computational experiments that underpin big science are hampered because the inordinate power draw of high-performance computing hardware makes the implementation of Exascale systems impractical. Likewise, current technologies are too energy-

inefficient to realize smaller and more intelligent wearable devices for a range of ubiquitous computing applications that can benefit society, such as personalized health-care.

Computing systems execute programs under the assumption that every instruction in a program is equally significant for the accuracy of the program output. This conservative approach to program execution may unnecessarily increase the energy footprint of software. Earlier work on approximate computing [1, 11, 15] shows that in several application domains, a program may produce virtually unaffected output if some parts of the program generate incorrect results or even fail completely. Many data-intensive applications and kernels from multimedia, data mining and visualization algorithms can tolerate a certain degree of imprecision.

As an example, Discrete Cosine Transform (DCT) is a module of popular video compression kernels, which transforms a block of image pixels to a block of frequency coefficients. DCT can be partitioned into different layers of significance, owing to the fact that the human eye is more sensitive to lower spatial frequencies, rather than higher ones. Then, by explicitly tagging operations that contribute to the computation of higher frequencies as less-significant, one can leverage smart underlying system software to trade-off video quality with energy and performance improvements.

Approximate computing is particularly interesting for programs that execute in energy-constrained environments. Consider for example an embedded system running on batteries, such as a mobile phone or an autonomous robot: when the battery is low, it may be preferable to run certain computations with a limited energy budget to prolong system lifetime, even if this comes at reduced output quality, or an acceptable compromise in user experience. As another example, cloud providers contemplate billing their clients based on the energy consumption of the hosted client applications. Clients would like to make their applications energy-aware and flexible, so that the energy cost of each application fits the owner's available budget. Furthermore, the willingness of a specific client to pay for energy may vary over time.

In this paper we introduce the first *significance-driven* programming framework for *energy-constrained approximate computing*. The framework comprises a programming model, a compilation-profiling-modeling toolchain and a runtime system. The programming model allows the developer to express the significance of computational tasks, depending on how strongly these tasks contribute to output quality. The developer can also provide approximate versions of selected tasks with lower complexity than that of their accurate counterparts. Approximate tasks may return inaccurate results or just a meaningful default value.

Our framework compiles and subjects each program to an offline profiling phase that uses different input data sets in order to measure the energy footprint of the program under different levels of concurrency, different processor frequency steps, and different degrees of approximation. This information is used to train a model, which is then employed, at runtime, to pick the proper configuration that achieves the highest output quality under a user-defined energy budget for new data sets.

This paper makes four contributions: (i) We introduce a programming model that allows the developer to structure the computation in terms of distinct tasks with different levels of significance, and to supply approximate versions of non-significant tasks; (ii) We introduce a profiling and model-training process to predict the energy footprint of programs as a function of the input size, the number and frequency configuration of the cores used to run the program and the ratio of tasks that are executed accurately; (iii) We introduce a runtime system that employs our model to pick the configuration that will achieve the highest possible output quality within a user-defined energy budget; (iv) We experimentally evaluate our approach for several application benchmarks, showing that our framework model performs very well in most cases, and achieves better output quality compared to loop perforation [17] (a well-known compiler-based approximation technique) for the same energy budget.

Specifically, our system can predict energy consumption accurately for all but three out of a total of nine benchmarks. This prediction is used effectively by the runtime system to degrade output quality in a graceful way, even when operating under severe energy constraints (down to 20% of the energy footprint of the most efficient accurate execution). In one of the three benchmarks where our model fails to make good predictions, application behavior depends not only on the size but also on the structure of the input data. The other two benchmarks have widely varying locality patterns, which in turn lead to additional data transfers between the last-level non-shared caches of the cores. Such inherently unpredictable programs are not amenable to profile-driven modeling and optimization.

The rest of the paper is structured as follows. Section 2 introduces the programming model. Sections 3 and 4 discuss the energy modeling and prediction methodology respectively, as well as the runtime system which exploits our model to allow graceful quality degradation under energy constraints. Section 5 presents the experimental evaluation of our framework on a multi core

server, using nine benchmarks that we ported to our programming model. Section 6 gives an overview of related work. Section 7 concludes the paper and presents directions for future work.

2 Programming Model

Part of the problem of energy inefficiency in computing systems is that all parts in a program are treated as equally important, despite the fact that only a subset of these parts may be critical to produce acceptable program output. Our vision is to elevate significance characterization as a first class concern in software development, similarly to parallelism and other algorithmic properties that programmers traditionally focus on. To this end, the main objectives of the proposed programming model are to enables programmers to: (i) express the significance of computations in terms of their contribution to the quality of the output; (ii) specify approximate alternatives for selected computations; (iii) express parallelism, beyond significance; (iv) optimizate and explore trade-offs, via offline and online methods.

We adopt a task-based paradigm where the programmer expresses both parallelism and significance using *#pragma* directives; this facilitates non-invasive and incremental code transformations without extensive code re-factoring and re-writing. Task scheduling decisions are taken by the runtime system, which considers resource availability and the data dependencies between tasks. The directives proposed by our model are extensions to those in the latest version of OpenMP [9]. Listing 1 illustrates Sobel filter, which we use as a running example, implemented with our programming model.

```
1  int sblX(byte *img, int y, int x) {
2    return img[(y-1)*WIDTH+x-1]
3      + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
4      - img[(y-1)*WIDTH+x+1]
5      - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
6  }
7
8  int sblX_appr(byte *img, int y, int x) {
9    return /* img[(y-1)*WIDTH+x-1]   Ommited taps */
10     + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
11     /* - img[(y-1)*WIDTH+x+1]   Ommited taps *//
12     - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
13 }
14
15 /* sblY and sblY_appr are similar */
16 void row_acc(byte *res, byte *img, int i) {
17   unsigned int p, j;
18   for (j=1; j<WIDTH-1; j++) {
19     p = sqrt(pow(sblX(img, i, j),2) +
20              pow(sblY(img, i, j),2));
21     res[i*WIDTH + j] = (p > 255) ? 255 : p;
22   }
23 }
24
25 void row_appr(byte *res, byte *img, int i) {
26   unsigned int p, j;
27   for (j=1; j<WIDTH-1; j++) {
28     /* abs instead of pow/sqrt,
29        approximate versions of sblX, sblY */
30     p = abs(sblX_appr(img, i, j) +
31             sblY_appr(img, i, j));
32     res[i*WIDTH + j] = (p > 255) ? 255 : p;
33   }
34 }
35
36 double sobel(void) {
37   int i;
38   byte img[WIDTH*HEIGHT], res[WIDTH*HEIGHT];
39   /* Initialize img array and reset res array */
40   ...
41   for (i=1; i<HEIGHT-1; i++)
42     #pragma omp task label(sobel) approxfun(row_appr) \
43       in(img[i*WIDTH+1:(i+1)*WIDTH-1]) \
44       out(res[i*WIDTH+1:(i+1)*WIDTH-1]) \
45       significant((i%9 + 1)/10.0)
46     row_acc(res, img, i); /* Compute a single
47                              output image row */
48   #pragma omp taskwait label(sobel) ratio(0.35)
49 }
```

Listing 1: Programming model use case: Sobel filter

Tasks are specified using the *#pragma omp task* directive (Listing 2), followed by the task body function. Task input and output is explicitly specified via the *in()* and *out()* clauses. This information is exploited by the runtime to detect task dependencies.

```
#pragma omp task [significant(...)] [label(...)]
    [in(...)] [out(...)] [approxfun(function())]
```

Listing 2: #pragma omp task

Task significance is given by the *significant()* clause. It takes values in the range [0.0, 1.0], indicating the relative importance of the task for the quality of the output. Depending on their significance, tasks may be approximated or dropped at runtime. The special values 1.0 and 0.0 are reserved for unconditional accurate and approximate execution, respectively.

For tasks with significance less than 1.0, the programmer may provide an alternative, approximate task body, through the *approxfun()* clause. This function is executed whenever the runtime opts to approximate a task. It typically implements a simpler version of the computation in the task, which may even degenerate to setting default values for the task output. If the runtime system decides to execute a task approximately and the programmer has not supplied an *approxfun* version, the task is dropped. The *approxfun* function implicitly takes the same arguments as the function implementing the accurate version of the task body.

Finally, *label()* can be used to group tasks under a common identifier (name), which is in turn used as a reference to implement synchronization at the granularity of task groups (discussed later in this section).

As an example, lines 41-46 of Listing 1 create a separate task to compute each row of the output image. The

significance of the tasks gradually ranges between 0.1 and 0.9 (line 45), so that there are no extreme quality fluctuations across the output image. The approximate function *row_appr* implements a lightweight version of the computation. All tasks created in the specific loop belong to the *sobel* task group, using *img* as input and *res* as output (lines 43-44).

```
#pragma omp taskwait [label(...)] [ratio(...)]
```

Listing 3: #pragma omp taskwait

Explicit barrier-like synchronization is supported via the *#pragma omp taskwait* directive (Listing 3). If the *label()* clause is missing, this serves as a global barrier, instructing the runtime to wait for all tasks spawned up to that point. Else, it becomes a barrier for the task group that is specified via *label()*, in which case the runtime system waits for the termination of all tasks of that group.

Importantly, *taskwait* can also be used to control the quality of application results. Using the *ratio()* clause, the programmer can instruct the runtime to execute in an accurate way (at least) the specified percentage of tasks (globally or within a group, depending on the scope of the barrier) while *respecting* task significance – a more significant task should not be executed approximately while a less significant task is executed accurately. The ratio takes values in the range [0.0, 1.0] and serves as a single, straightforward knob to *enforce* a minimum quality in the performance / quality / energy optimization space. Smaller ratios give the runtime more energy reduction opportunities, but with a potential penalty in terms of output quality.

As an example, line 48 of Listing 1 specifies a barrier for the tasks of the *sobel* task group. In this case the runtime is instructed to ensure that, at a minimum, the most significant 35% task of the group are executed accurately. Note that the runtime may opt for a higher ratio, e.g., if this is feasible with the energy budget of the program.

The programming model is implemented by a source-to-source compiler, based on the SCOOP [23] infrastructure. It recognizes the pragmas of the programming model, and lowers them to corresponding calls of the runtime system (discussed in Section 4). The resulting code is then compiled by the standard *gcc* tool-chain to produce the final executable.

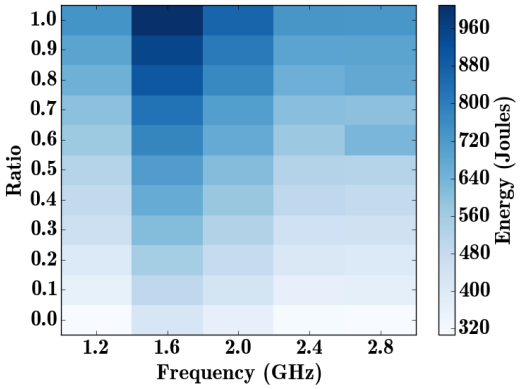

Fig. 1: Energy footprint of the *Fisheye* benchmark under different $(CPUFrequency, TaskRatio)$ configurations.

3 Modeling and Prediction of Application Energy Footprint

We introduce an analytical model to predict the energy consumption of an application under different input sizes and execution configurations, in terms of number of cores used, processor frequency, and the mix of accurately and approximately executed tasks. The reason for introducing the processor frequency as one of the parameters that are explored by our model is because we have experimentally observed that the energy footprint of a computation may correlate to the combination of frequency and the task approximation ratio in a non-trivial way.

For example, Figure 1 depicts the energy consumption of the *fisheye* benchmark (discussed in more detail in Section 5.1) for 16 cores, different CPU frequencies, and different ratios of accurate/approximate tasks. The plot shows that the most energy-efficient executions when low quality can be tolerated (ratios 0.0-0.3) are at either 1.2 or 2.8 GHz, while the best frequency when targeting higher quality (ratios 0.6-0.8) is 2.4 GHz. Also note that 1.6 GHz is a bad choice, independently of the desired quality of the end result.

Next we describe our modelling and prediction approach in more detail. As an underlying platform, we assume a general-purpose shared-memory architecture with multiple multi-core processors/CPUs. All cores within each CPU share the same last level cache and operate at the same frequency (as is the case with the popular Intel processors). We start by presenting the analytical model for the execution time of a multi-tasking computation on top of such a platform, and the energy that is expected to be consumed by it. We then discuss the process that is followed to train the model through an offline profiling and fitting phase.

3.1 Analytical Model of Execution Time

Let a computation employ m task-groups, with each group i consisting of n_i tasks. Let the accurate task ratio for group i be r_i. Also, let the average execution time of accurate and approximate task versions for group i be equal to $\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$, respectively. For simplicity, we assume that a task group is well-balanced, and that all tasks roughly take the same time to execute, subject only to whether they are executed accurately or approximately. Then, the time that is required for the computation to be executed in a purely sequential way, is given by Equation 1, as a function of the input size s, the CPU frequency f, the ratios, and number of tasks for each group.

$$T_{seq}(f, \mathbf{r}, s, \mathbf{n}) = \sum_{i=1}^{m} \Big(n_i \cdot (r_i \cdot \overline{T_{accurate_i}(f, s, n_i)} + (1 - r_i) \cdot \overline{T_{approx_i}(f, s, n_i)}) \Big) \quad (1)$$

Note that larger problem sizes s may also require a larger number of tasks in certain groups or more work per task or both. Indeed, the number of tasks n_i and the time it takes for a task of group i to execute in its accurate or approximate version ($\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$) are open parameters of the model. This makes it possible to implicitly account for effects that can significantly affect task execution time, such as locality, caching and memory traffic due to different input and intermediate data footprints associated with different problem sizes.

Equation 2 estimates the parallel execution time for the same computation, as a function of the number of cores c that are used. The assumption is that all cores run at the same frequency f, which is typically the case in most off-the-shelf platforms, including the one we use in our evaluation.

$$T_{par}(f, \mathbf{r}, s, \mathbf{n}, c) = \frac{T_{seq}(f, \mathbf{r}, s)}{c \cdot scaling(f, s, c)} \quad (2)$$

The term $scaling(f, s, c)$ captures the scalability of the computation as a function of input size s, the frequency f at which (all) cores run, and the number of cores c. On a a multiprocessor with multi-core CPUs, we assume a "packed" CPU allocation strategy, whereby the runtime exploits all cores in a given CPU before using the cores in another CPU. Thus, at most one CPU can have unused cores, which is the most energy efficient allocation strategy for common platforms.

3.2 Analytical Model of Power Consumption

The power consumption of the processing elements is given in Equation 3.

$$P = P_{background}(f, c) + P_{dynamic}(f, c, s, \mathbf{r}) \quad (3)$$

The $P_{background}$ component captures the "background" power consumed by the number of active cores c running at frequency f, when idle. The $P_{dynamic}$ component corresponds to the "dynamic" power consumption, which depends on the computation that is actually being executed. This in turn is a function of the number of cores used, the frequency of these cores, the input size and the mix of accurate/approximate tasks. The rationale behind this that the same task-group might behave differently for different values of *ratio*. The actual accurate/approximate mix affects the instruction mix of the overall application as well as the memory locality and access pattern.

3.3 Offline Profiling and Model Fitting

In a profiling phase, the computation is executed with three different, representative input data-sets, of varying size s (and thus also different memory footprints). To account for locality, caching and memory traffic effects, we execute with a small working set that fits in the last level cache (LLC) of a single processor, a large working set that exceeds the total LLC capacity of all processors in the system[1] and, finally, an intermediate working set. For each input, we execute the computation for all possible configurations (varying the number of cores c, the frequency f and the task ratio $\mathbf{r}$). We measure the average execution time of approximate and accurate tasks for each task group, and the total execution time of each group.

Then follows a step-wise model fitting phase, where the performance data that was gathered in the profiling phase is used as input to a regression process. The objective is to train the different terms of the analytical models presented above, so that they predict execution time and energy consumption of a given computation for the different configurations.

The first step is to produce estimation functions for $\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$ in Equation 1. We perform regression to map the average execution time of tasks in a given group i, for both their approximate and accurate versions, to the frequency f, problem size s, and number of tasks n_i. A separate function is created for each of

[1] We skip problem sizes which are unrealistic. This is done for the large data-set in the Monte Carlo and MD benchmarks.

the frequencies that are supported by the platform. We use the average execution time of tasks that is observed when executing across all ratios. Exponential, polynomial and linear fitting functions are all attempted, and we use the one which minimizes the prediction error with respect to profiling data.

Next, we produce the function for the *scaling* term in Equation 2, using the measured sequential and parallel execution times for different combinations of problem sizes and number of cores (the latter for parallel execution times). We also experiment with exponential, polynomial and linear fitting functions. The result is a separate function for each frequency, which correlates scalability to problem size and the number of cores used.

In a last step, a similar approach is followed to produce the function for the dynamic power consumption $P_{dynamic}$ component used in Equation 3. Again, a separate function is produced for each frequency, which returns an estimation based on the problem size, the task ratio and the number of cores used. Note that $P_{background}$ can be computed just once, measuring the power consumption as a function of the number of cores that are turned on, without running any computation.

The whole profiling and model-fitting process is repeated for each different application, yielding different functions for each case. This application-specific information is then made available to the runtime system in order to pick the best configuration for a given energy budget.

4 Runtime System

We extend a task-based parallel runtime system that implements OpenMP 4.0-style task dependencies [20] to support our programming model for energy-aware computing.

The runtime system implements a master/slave work sharing scheduler. The master thread starts executing the main program sequentially. Spawned tasks are distributed to local, per-core work queues round-robin. Tasks are released for execution when their true dependencies are satisfied. The runtime system implements an efficient mechanism for identifying and enforcing dependencies between tasks that arise from annotations of the side effects of tasks with *in(...)* and *out(...)* clauses. A ready for execution task moves from a local work queue to a local ready queue. Workers select the oldest tasks from their ready queues for execution. Work stealing is used to facilitate load balancing between workers.

The main objective of the energy-aware runtime system is to execute the application within the energy budget specified by the user, while achieving the highest possible output quality. Energy budgets can be defined either relatively to the energy consumption of the most energy-efficient fully accurate execution, or as an absolute value. The energy budget is set with an environment variable (*ENERGY_BUDGET*). Given the energy budget, the problem size and the number of created tasks, the runtime system uses the offline-trained model to predict the Pareto-optimal configuration in terms of number of cores, processor frequency, and ratio of accurate/approximate tasks. This configuration is selected to achieve execution within the energy budget, while maximizing the ratio of accurate tasks. If the runtime cannot identify an execution configuration within the requested energy budget, it opts to execute with the least energy consuming configuration.

Beyond achieving the selected ratio of accurate/approximate tasks and staying within the energy budget, the runtime system also has to respect user-provided wisdom on the relative importance of tasks for output quality: high significance tasks should have higher priority for accurate execution over lower significance tasks in the same task group.

Ideally, the runtime system can have *a priori* information on the number of tasks to be issued in a task group and the distribution of significance levels within the group. In this case it is straightforward to execute approximately the tasks with the lowest significance in each group in order to achieve the target ratio. If this is not the case, the respective information has to be collected at runtime. We accomplish this by having the master thread buffering tasks on creation, while postponing task issuing to worker queues. When the buffer is full, or when a synchronization construct is reached, the tasks in the buffer are sorted by significance. Then, the runtime estimates the optimal execution configuration and tags each task for accurate or approximate execution according to its relative significance and the target ratio. We use two runtime system algorithms, one using global state for preserving the exact accurate task ratio and one using distributed local state for estimating the accurate task ratio from partial execution-time information. The algorithms are presented in an earlier paper [21].

5 Experimental Evaluation

We use nine benchmarks to validate our framework and its ability to execute applications with a pre-defined energy budget, while gracefully trading off output quality with energy efficiency. The benchmarks have been manually ported to the proposed significance-driven programming model. We compare our framework against

loop perforation [17] in terms of quality of results under the same energy constraints.

5.1 Benchmarks

We apply different approximation approaches to each benchmark, subject to algorithmic characteristics of the underlying computation.

Sobel is a 2D filter for edge detection in images. The approximate version of the tasks uses a lightweight Sobel stencil with just 2/3 of the filter taps. Additionally, it substitutes the costly formula $\sqrt{{sbl_x}^2 + sbl_y^2}$ with its approximate counterpart $|sbl_x| + |sbl_y|$. Significance is assigned to tasks in a round-robin manner, which ensures that approximated pixels are uniformly distributed throughout the output.

Discrete Cosine Transform (DCT) is a module of the JPEG compression and decompression algorithm [18]. We assign higher significance to tasks that compute lower frequency coefficients, as the human eye is more sensitive to those frequencies. Should a task be executed approximately, the computation is dropped.

Fisheye lens distortion correction [2] is an image processing application which transforms images distorted by a fisheye lens back to the natural-looking perspective space. The exact algorithm initially associates pixels of the output, perspective space image, to points in the distorted image. Then, interpolation on a 4×4 window is applied to calculate each pixel value of the output, based on the values of neighboring pixels of the corresponding point in the distorted image. The approximate task also performs the inverse mapping procedure, however instead of calculating each output pixel by interpolating around the corresponding point in the input, it simply uses the value of the nearest neighboring pixel.

K-means is an iterative algorithm for grouping data points from a multi-dimensional space into k clusters. Each iteration consists of two phases: Chunks of data points are first assigned to different tasks, which independently determine the nearest cluster for each data point. Then, another task group is used to update the cluster centers by taking into account the position of the points that have moved. The first phase is characterized as non-significant, because errors in the assignment of individual points to clusters can be tolerated. Approximate tasks compute a simpler version of the Euclidean distance while also considering only half of the total dimensions. The second phase is significant, as it is harder to recover from a wrong estimate of a cluster center.

MC [22] applies a Monte Carlo approach to estimate the boundary of a sub-domain within a larger partial differential equation (PDE) domain, by performing random walks from points of the sub-domain boundary to the boundary of the initial domain. Approximate configurations drop a percentage of the random walks and the corresponding computations. An approximate, lightweight methodology is also used to decide how far from the current location the next step of a random walk should move.

Canneal, a code from the PARSEC benchmark suite [3], applies an annealing methodology to optimize the routing cost of a chip design. This optimization method pseudo-randomly swaps net-list elements. If the swap results in better routing cost it is accepted immediately. Local minima are avoided by rarely accepting swaps that increase the routing cost of the net-list. Approximate tasks try less swaps (1/8) than accurate ones. All tasks are assigned the same significance value, so the tasks to be approximated are randomly selected by the runtime, according to the target $r$atio.

The *MD* (molecular dynamics) application simulates the kinematic behaviour (position and velocity) of liquid Argon atoms within a bounded space, under the effects of a force produced by a Lennard-Jones pair potential [5]. The potential is defined as a function of distance (r) and two material specific constants (σ and ϵ):

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{4}$$

The significance of the interaction between atoms is strongly correlated with the distance between them. The greater the distance between atom A and atom B, the less the kinematic properties of A affect those of B (and vice versa). In the task-based version of *MD*, the 3D container of the particles is partitioned into regions which are updated every few time-steps to populate a list of the particles that reside inside them. For each given atom, one task per region is instantiated to calculate the forces that operate on the atom due to the particles contained in that specific region. The task that performs the calculation for the region that contains the atom in question, is tagged as fully significant. The significance of tasks that are responsible for other regions drops with increasing distance to the atoms home region.

BlackScholes is a benchmark of the Parsec suite [3]. It implements a mathematical model for a market of derivatives, which calculates the buying and selling of assets so as to reduce the financial risk. The computation of a stock price can be broken down to 4 blocks of code A, B, C, D, with $sig(A) > sig(B) \gg sig(C) > sig(D)$. The least important parts (C and D) are approximated using less accurate but faster implementa-

tions of mathematical functions such as *exp* and *sqrt* [7].

There is a wide variety of applications which model the behavior of materials when colliding or being subject to forces. *Lulesh* [6] implements a solution of the Sedov blast problem for a material in three dimensions. It defines a discrete mesh that covers the region of interest and it partitions the problem into a collection of elements where hydrodynamic equations are applied. We introduce an approximate version of the hourglass force calculation. Similarly to *MD* we consider the significance of particles to be diminishing when moving away from the impact site. Computations involving the least significant particles can be dropped at execution time.

5.2 Experimental Methodology

The experimental analysis was carried out on a system equipped with two Intel(R) Xeon(R) E5-2650 processor, and 64 GB shared DRAM. Each processor has 8 cores and can be clocked at 1.2, 1.6, 2.0, 2.4, or 2.8GHz. Energy and power are measured using the Running Average Power Limit (RAPL) registers of the processors.

The profiling phase uses a pool of representative input sets for each benchmark, discussed in Section 3. At the end of the profiling and model fitting process, each benchmark is associated with a model estimating its energy consumption according to the input size and execution configuration. This formula is, in turn, used by the runtime system to take online decisions on the execution configuration.

To evaluate our approach, we use for all benchmarks unseen input sets (and input set sizes) which have not been used during the training phase. All benchmarks are executed accurately, in all possible core and frequency configurations. From those executions we identify the one that consumes the least energy. This is our baseline scenario for each benchmark.

We then perform a number of experiments for each benchmark, while requesting a gradually smaller energy budget, expressed as a percentage of the baseline. The framework uses the model to decide, at runtime, the ratio, and concurrency level with which it can achieve execution within the requested energy budget, while minimizing the impact on output quality by maximizing the ratio of accurate tasks.

We present a comparison of the quality achieved using our framework with a perforated execution of each benchmark targeting the same energy budget. We also present the optimal (oracular) configuration (cores, ratio) for each case and compare it to the one selected by our system.

5.3 Experimental Evaluation and Discussion

Figures 2 summarizes our results. In all charts the horizontal axis represents the requested energy budget, as a percentage of the energy consumed by the most energy-efficient accurate execution. The Y-axis of the first set of charts corresponds to the energy that was actually consumed by approximate executions as a percentage of the energy consumed by the accurate execution. The second set of charts is used to quantify output quality. We compare our framework against an oracle (optimal) configurator, as well as against loop perforation for which an oracle has selected the optimal number of dropped iterations.

For the first three applications *(DCT, Sobel, Fisheye)* output quality is quantified using *PSNR* (higher is better). PSNR is a logarithmic metric. For *Kmeans*, the metric of the quality of output is the relative difference of the average distance between data points and the center of the cluster they are assigned to, compared with that of the fully accurate execution (lower is better). For the remaining five benchmarks we report the relative error with respect to an accurate execution (lower is better).

Our framework produces program configurations which result in energy consumption that is very similar to the optimal. Even in cases when the runtime opts for a non-optimal configuration, the difference in the achieved energy footprint and quality of results is negligible, with the exception of Canneal, Kmeans, and Lulesh which are discussed in more detail later in this section. Both our approach and the optimal tend to adapt concurrency to utilize all cores of both CPUs. This is expected, as the dominant term in power estimation is due to the activation of additional CPUs.

Imaging and media applications are well-suited for our programming framework, as they take full advantage of the significance and approximation features of the programming model. Moreover, the specific implementations scale to larger inputs by adapting the number of tasks, instead of modifying the work per task. Therefore it is easier for our model to predict their behavior with high accuracy. Finally, the execution of approximate tasks has a straightforward and easy to model effect on execution time: more approximate tasks result in less computation and thus more energy savings.

Sobel DCT, and *Fisheye* can execute with as little as 50% of the energy required by the optimum accurate execution and match the quality achieved by the oracle. The minimum energy required depends mainly on the complexity of the approximation function we use. At the same time, the complexity and sophistication

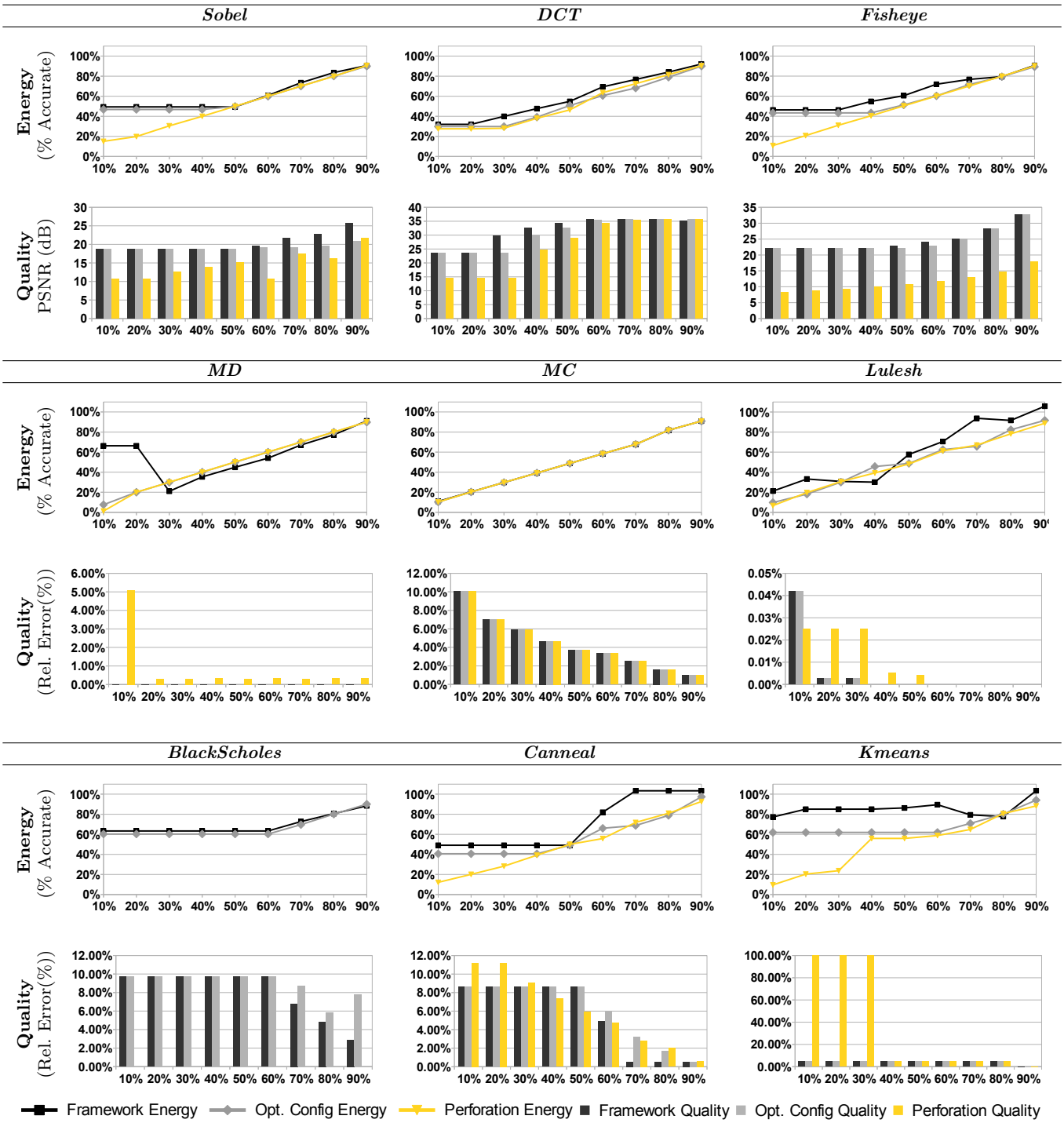

Fig. 2: Quality and energy metrics for different energy targets (as a percentage of the most energy-efficient accurate execution). Energy & quality plots show the results achieved by our system, an oracle selecting the optimal configuration and loop perforation.

of the approximation function determines output quality for the most aggressive degrees of approximation. When approximating all tasks we observe PSNRs equal to 18.70, 23.64, 22.09 dB for *Sobel*, *DCT*, and *Fisheye* respectively. Perforated executions capped at the same amount of energy produce results of inferior quality, corresponding to PSNRs of 10.75, 14.48 and 8.19 dB respectively. Our methodology clearly results in higher quality of results with the same energy budget. However it sometimes slightly overshoots the energy budget con-

Fig. 3: Lena portrait compressed and decompressed using *DCT* with a ratio of 0.3.

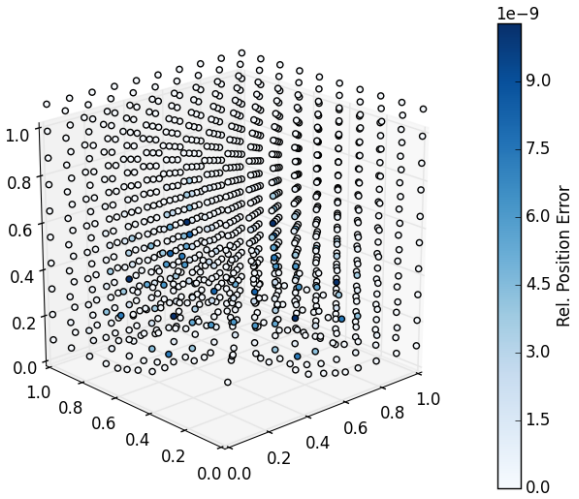

Fig. 4: Final positions of particles for an approximate execution with ratio 0.2. Particles have been colored according to the relative error of their position with respect to an accurate execution.

straints by picking ratio values which are higher than the optimal. In the case of *DCT* we overspend, on average, by 6.2%, for *Sobel* this number is 2.1% and finally *Fisheye* overspends by 6.4%. This leads to a pitfall in Figure 2 where our framework seems to outperform the oracle, which is clearly not possible. Figure 3 depicts the Lena portrait compressed and decompressed using *DCT* with a ratio of 0.3. The resulting output has a PSNR of 34.62 dB (no visible quality loss), at a 45% energy gain with respect to the most energy efficient accurate execution.

MD is another well-behaved application for our framework. In most cases we choose configurations which result in energy consumption that is very close to what an oracle achieves. In fact, our estimations, excluding the energy budgets 10% and 20% result in energy consumption which differs by 4.5% from the user specified energy budget. Moreover, we always achieve a better quality of results than the perforated version of the benchmark. With just 30% of the energy budget of the most energy efficient accurate execution *MD* computes results with a relative error in the order of 0.0006%.

For *MC* we observe that our framework makes optimal choices in almost every case. Approximation in *MC* drops random walks, similarly to perforation, therefore we observe similar results with both techniques. A lower energy budget results in pruning some of the random walks of the search space. This reduces energy, albeit with a measurable impact in quality. We can achieve consumption as low as 30% of the energy required by the most energy-efficient accurate execution, using a ratio of 0.2 which results in a relative error of 5.9%.

Regarding *Lulesh*, we notice that for energy budgets higher than 10% the framework version always produces higher quality than the perforated one, but it tends to overshoot the energy budget. The case of the energy budget being 10% of the optimal accurate is particularly peculiar: the perforated version is better in terms of quality and energy than both our framework and the oracle. This is due to the fact that the approximated executions have to spend some of their energy budget to compute the significance of tasks. Furthermore, approximated tasks do not access the memory with a regular pattern. Elements are visited according to their distance from the point of blast. Unfortunately, this access pattern affects memory locality in a detrimental fashion. On the other hand the perforated version has a regular memory access pattern and the respective energy drops linearly with respect to the number of the dropped iterations. However, for higher – and realistic – energy budgets our approach always produces results of better quality compared with the perforated executions. We do have to note that the two issues described above limit the accuracy of our framework's estimations. Figure 4 depicts the positions of particles calculated by a small-scale approximate execution with ratio set to 0.2. Particles are colored according to the relative error of their final position with respect to the fully accurate execution. The maximum relative error is negligible (in the order of 10^{-8}).

BlackScholes calculates prices for a number of assets. The main loop iterates across different assets, however there is no loop involved in the calculation of each particular asset. As a result, perforation is not applicable and we limit our comparison between the proposed framework and the optimal configuration by an oracle. Because of the computational cost of approximate tasks, the lowest energy budget obtained by the Oracle is 60% of the accurate execution; our framework fol-

lows closely at 63.3%. Once again, we produce results of higher quality than the oracle for energy budgets higher than 60% due to slightly overshooting the target energy budget by executing more accurate tasks.

Our model is less accurate in its predictions for *Canneal*. This is a consequence of the bad, unpredictable locality pattern if the application. Canneal uses large data structures to store information on net-list elements. The random way each task accesses memory locations increases cache misses, in particular false sharing misses that introduce excessive data transfers between the last-level non-shared caches of cores. This unpredictable behavior cannot be modeled accurately by our framework. As a result, we underestimate the execution time of the application and often select configurations that do not satisfy the energy constraints.

Kmeans reveals the limitations of our approach. It can not be modeled effectively, as it is iterative, with the number of iterations being heavily dependent on the characteristics of the input set (and not just the input set size). Moreover, wrong decisions in the approximate tasks (point classification) tend to increase point movement between clusters, and thus the workload of accurate tasks (cluster center calculation). In addition, even when we approximate 100% of the point classification tasks, we can only reduce the energy footprint by at most 60% because our approximation disregards half of the coordinates of each point. For such applications, a blind approach such as loop perforation proves to be a viable solution for medium-to-large energy budgets as it produces solutions which are as good as our framework using less energy.

To sum up the results of our experimental campaign we note that there are scenarios in which it simply impossible to arbitrarily decrease the energy footprint of an application due to the fact that even the approximate versions of tasks come with computational cost. We do observe however, that in the bulk of the test-cases our framework succeeds in gracefully trading quality to reduce the cost of executing an application.

6 Related Work

To the best of our knowledge this work is the first to propose a parallel programming model for significance-aware approximate computation, and the first to model and explore a design space for approximate parallel applications that achieves quality optimization under resource constraints. Our work departs from prior art in approximate computing in several ways.

6.1 Parallel Approximation Frameworks

Quickstep [8], is a tool that approximately parallelizes sequential programs. The parallelized programs are subjected to statistical accuracy tests for correctness. Quickstep tolerates races that occur after removing synchronization operations that would otherwise be necessary to preserve the semantics of the sequential program. Quickstep thus exposes additional parallelization and optimization opportunities via approximating the data and control dependencies in a program. On the other hand, QuickStep does not enable algorithmic and application-specific approximation, which is the focus of our work. and does not include energy-aware optimizations in the runtime system.

Variability-aware OpenMP [11] and variation tolerant Open-MP [10], are a sets of OpenMP extensions that enable a programmer to specify blocks of code that can be computed approximately. The programmer may also specify error tolerance in terms of the number of most significant bits in a variable which are guaranteed to be correct. We follow a different scheme that allows approximate –in our context, not significant– tasks to be selectively dropped from execution and dynamic error checks to detect and recover from errors via selective task restarting. Variability-aware OpenMP applies approximation only to specific FPU operations, which execute on specialized FPUs with configurable accuracy. Our framework applies selective approximation at the granularity of tasks, using the significance abstraction. Our programming and execution model thus provides additional flexibility to drop or approximate code, while preserving output quality. Furthermore, our framework does not require specialized hardware support and runs on commodity systems.

6.2 Other Approximation Frameworks

Several frameworks for approximate computing discard parts of code at runtime, while asserting that the quality of the result complies with quality criteria provided by the programmer. Green [1] is an API for loop-level and function approximation. Loops are approximated with a reduction of the loop trip count. Functions are approximated with multi-versioning. The API includes calibration functions that build application-specific QoS models for the outputs of the approximated blocks of code, as well as re-calibration functions for correcting unacceptable errors that may incur due to approximation. Sloan et al. [19] provide guidelines for manual control of approximate computation and error checking in software. These frameworks delegate the control of approximate code execution to the programmer. We ex-

plore an alternative approach where the programmer uses a higher level of abstraction for approximation, namely computational significance, while the system software translates this abstraction into energy- and performance-efficient approximate execution.

Loop perforation [17] is a compiler technique that classifies loop iterations into critical and non-critical ones. The latter can be dropped, as long as the results of the loop are acceptable from a quality standpoint. Input sampling and code versioning [25] also use the compiler to selectively discard inputs to functions and substitute accurate function implementations with approximate ones. Similarly to loop perforation and code versioning, our framework benefits from task dropping and the execution of approximate versions of tasks. However, we follow a different approach whereby these optimizations are driven from user input on the relative significance of code blocks and are used selectively in the runtime system to meet user-defined quality criteria energy savings and performance gain. While these approaches demonstrate aggressive performance optimization thanks to approximation, they do not consider parallelism in execution. Furthermore, these techniques operate at a granularity different than parallel tasks or specific runtime energy optimization opportunities which are exposed through approximation.

Several software and hardware schemes for approximate computing follow a domain-specific approach. ApproxIt [24] is a framework for approximate iterative methods, based on a lightweight quality control mechanism. Unlike our task-based approach, ApproxIt uses coarse-grain approximation at a minimum granularity of one solver iteration. Gschwandtner et al. use a similar iterative approach to execute error-tolerant solvers on processors that operate with near-threshold voltage (NTC) and reduce energy consumption by replacing cores operating at nominal voltage with NTC cores [4]. Schmoll et al. [16] present algorithmic and static analysis techniques to detect variables that must be computed reliably and variables that can be computed approximately in an H.264 video decoder. Although we follow a domain-agnostic approach in our approximate computing framework, we provide sufficient abstractions for implementing the aforementioned application-specific approximation methods.

Other tools automate the generation and execution of approximate computations. SAGE [14] is a compiler and runtime environment for automatic generation of approximate kernels in machine learning and image processing applications. Paraprox [13] implements transparent approximation for data-parallel programs by recognizing common algorithmic kernels and replacing them with approximate equivalents. ASAC [12] provides sensitivity analysis for automatically generated code annotations that quantify significance. We do not explore automatic generation of approximate code in this work. However, our techniques for quality-aware, selective execution of approximate code are directly applicable to scenarios where the approximate code is derived from a compiler, instead of source code annotations.

7 Conclusion

This paper introduced a directive-based programming model that allows developers to specify computational significance at the granularity of tasks. This information is used to achieve energy-constrained execution with graceful quality degradation. An offline, profile-based, training process produces a model which predicts the energy footprint of a given application as a function of its input size, the number of cores used, the processor frequency, and the ratio of accurate to total number of tasks. This model is exploited by the runtime system of an energy-constrained multi-core platform to steer execution towards a configuration that maximizes quality of output while complying with energy constraints.

The experimental evaluation across several benchmark codes shows that the exploitation of programmer wisdom on the significance of computations is necessary in order to achieve energy constrained execution without excessive quality loss. This is particularly evident when comparing our approach against loop perforation [17], a blind approximation technique applied at the compiler level. In this work we consider programmer wisdom as the corner stone of significance-driven computing. However, our intuition indicates that an automatic, or at least semi-automatic significance analysis of computations may be realistic and would extend the applicability of the proposed framework.

In the future, we plan to investigate automatic significance analysis methods. We also intend to explore alternative optimization scenarios, by combining profile-based methodologies with dynamic heuristics in the runtime system. Moreover, we will investigate effective domain-specific ways to express quality constraints, and use the framework to achieve automated energy-efficient execution within quality limitations. Finally we plan to work on cost effective ways to evaluate the intermediate quality of results at runtime.

Acknowledgements This work has been partially supported by: (a) The European Commission's 7th Framework Programme (FP7/2007- 2013) under grant agreements FP7-323872 (Project "SCoRPiO") and FP7-327744 (NovoSoft, Marie Curie Actions), (b) The "Aristeia II" action (grant agreement 5211, project "Centaurus" of the operational program Education

and Lifelong Learning which is co-funded by the European Social Fund and Greek national resources, and (c) The UK Engineering and Physical Sciences Research Council under grant agreement EP/L000055/1 (ALEA).

References

1. W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.
2. N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier. Real-time fisheye lens distortion correction using automatically generated streaming accelerators. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 149–156. IEEE, 2009.
3. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.
4. P. Gschwandtner, C. Chalios, D. Nikolopoulos, H. Vandierendonck, and T. Fahringer. On the potential of significance-driven execution for energy-aware hpc. *Computer Science - Research and Development*, pages 1–10, 2014.
5. J. E. Jones. On the Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 106(738):441–462, 1924.
6. I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.
7. P. Mineiro. fastapprox. `http://code.google.com/p/fastapprox/`, 2012.
8. S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.*, 12(2s):88:1–88:26, May 2013.
9. OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.
10. A. Rahimi, A. Marongiu, P. Burgio, R. K. Gupta, and L. Benini. Variation-tolerant openmp tasking on tightly-coupled processor clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 541–546, San Jose, CA, USA, 2013. EDA Consortium.
11. A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 35:1–35:10, Piscataway, NJ, USA, 2013. IEEE Press.
12. P. Roy, R. Ray, C. Wang, and W. F. Wong. Asac: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 95–104, New York, NY, USA, 2014. ACM.
13. M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.
14. M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.
15. 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 *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.
16. F. Schmoll, A. Heinig, P. Marwedel, and M. Engel. Improving the fault resilience of an h.264 decoder using static analysis methods. *ACM Trans. Embed. Comput. Syst.*, 13(1s):31:1–31:27, Dec. 2013.
17. S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. 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*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.
18. A. Skodras, C. Christopoulos, and T. Ebrahimi. The jpeg 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36–58, Sept. 2001.
19. J. Sloan, J. Sartori, and R. Kumar. On software design for stochastic processors. In *Proceedings of the 49th Annual Design Automation Conference*, DAC

'12, pages 918–923, New York, NY, USA, 2012. ACM.

20. G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Bddt: Block-level dynamic dependence analysis for task-based parallelism. In *Advanced Parallel Processing Technologies*, pages 17–31, 2013.
21. 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. *CoRR*, abs/1412.5150, 2014.
22. M. Vavalis and G. Sarailidis. Hybrid-numerical-PDE-solvers: Hybrid Elliptic PDE Solvers. `http://dx.doi.org/10.5281/zenodo.11691`, Sep 2014.
23. F. S. Zakkak, D. Chasapis, P. Pratikakis, A. Bilas, and D. S. Nikolopoulos. Inference and declaration of independence: Impact on deterministic task parallelism. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 453–454, New York, NY, USA, 2012. ACM.
24. Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 97:1–97:6, New York, NY, USA, 2014. ACM.
25. Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM.