A Significance-Driven Programming Framework for Energy-Constrained Approximate Computing

Vassiliadis, V., Chalios, C., Parasyris, K., Antonopoulos, C. D., Lalis, S., Bellas, N., ... Nikolopoulos, D. (2015). A Significance-Driven Programming Framework for Energy-Constrained Approximate Computing. In Proceedings of the ACM International Conference on Computing Frontiers (CF). [9] ACM. DOI: 10.1145/2742854.2742857

Published in:
Proceedings of the ACM International Conference on Computing Frontiers (CF)

Document Version:
Peer reviewed version

A Significance-Driven Programming Framework for Energy-Constrained Approximate Computing

Vassilis Vassiliadis[*] vasiliad@inf.uth.gr
Charalampos Chalios[†] cchalios01@qub.ac.uk
Konstantinos Parasyris[*] koparasy@inf.uth.gr
Christos D.Antonopoulos[*] cda@inf.uth.gr
Spyros Lalis[*] lalis@inf.uth.gr
Nikolaos Bellas[*] nbellas@inf.uth.gr
Hans Vandierendonck[†] h.vandierendonck@qub.ac.uk
Dimitrios S.Nikolopoulos[†] d.nikolopoulos@qub.ac.uk

[*]Centre for Research & Technology Hellas Greece
[*]University of Thessaly Greece
[†]Queen's University Belfast United Kingdom

October 28, 2015

Abstract

Approximate execution is a viable technique for energy-constrained environments, provided that applications have the mechanisms to produce outputs of the highest possible quality within the given energy budget.

We introduce a framework for energy-constrained execution with controlled and graceful quality loss. A simple programming model allows users to express the relative importance of computations for the quality of the end result, as well as minimum quality requirements. The significance-aware runtime system uses an application-specific analytical energy model to identify the degree of concurrency and approximation that maximizes quality while meeting user-specified energy constraints.

Evaluation on a dual-socket 8-core server shows that the proposed framework predicts the optimal configuration with high accuracy, enabling energy-constrained executions that result in significantly higher quality compared to loop perforation, a compiler approximation technique.

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 healthcare.

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 a system. Earlier work on approximate computing [1, 8, 11] 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.

Approximate computing can benefit 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 default input data sets to measure the program energy footprint under different degrees of parallelism and approximation. This information is used to train a model that estimates the energy footprint of the program for unseen inputs and execution configurations, where parallelism and degree of approximation may vary. At runtime, the energy estimation model identifies the configuration that achieves the highest output quality under a user-defined energy budget.

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 of cores used to run the program and the ratio of program tasks that are executed accurately; (iii) We introduce a runtime system that uses our model to enforce execution configurations with the highest possible output quality within a user-defined energy budget; (iv) We experimentally evaluate our approach and show that it is superior to loop perforation [13], a compiler-based approximation technique.

Our system can predict energy consumption accurately for all but one of our benchmarks. This prediction is used effectively by the runtime system to achieve up to 80% energy savings, while gracefully degrading output quality. In the one problematic benchmark, execution time and energy depend not only on input size and execution configuration, but also on the structure of the input data. Such a program is 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 multicore server, using six benchmark kernels 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 allow 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) enable optimization and exploration of trade-offs, via offline and online methods.

```
int sblX(byte *img, int y, int x) {
  return img[(y-1)*WIDTH+x-1]
    + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
    - img[(y-1)*WIDTH+x+1]
    - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
}

int sblX_appr(byte *img, int y, int x) {
  return /* img[(y-1)*WIDTH+x-1]   Ommited taps */
    + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
    /* - img[(y-1)*WIDTH+x+1]   Ommited taps *//
    - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
}

/* sblY and sblY_appr are similar */
void row_acc(byte *res, byte *img, int i) {
  unsigned int p, j;
  for (j=1; j<WIDTH-1; j++) {
    p = sqrt(pow(sblX(img, i, j),2) +
             pow(sblY(img, i, j),2));
    res[i*WIDTH + j] = (p > 255) ? 255 : p;
  }
}

void row_appr(byte *res, byte *img, int i) {
  unsigned int p, j;
  for (j=1; j<WIDTH-1; j++) {
    /* abs instead of pow/sqrt,
       approximate versions of sblX, sblY */
    p = abs(sblX_appr(img, i, j) +
            sblY_appr(img, i, j));
    res[i*WIDTH + j] = (p > 255) ? 255 : p;
  }
}

double sobel(void) {
  int i;
  byte img[WIDTH*HEIGHT], res[WIDTH*HEIGHT];
  /* Initialize img array and reset res array */
  ...
  for (i=1; i<HEIGHT-1; i++)
    #pragma omp task label(sobel) approxfun(row_appr) \
      in(img[i*WIDTH:(i+1)*WIDTH-1]) \
      out(res[i*WIDTH:(i+1)*WIDTH-1]) \
      significant((i%9 + 1)/10.0)
    row_acc(res, img, i); /* Compute a single
                             output image row */
  #pragma omp taskwait label(sobel) ratio(0.35)
}
```

Listing 1: Programming model use case: Sobel filter

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. Our model uses the latest version of OpenMP [6], so task scheduling decisions are taken by the runtime system, which also considers the data dependencies between tasks. Listing 1 depicts our programming model in the implementation of the Sobel filter, which we use as a running example.

Tasks are specified using the *#pragma omp task* directive (Listing 2), followed by the task body function. The significance of each task 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 used for tasks that must be *unconditionally* executed accurately and approximately, respectively.

For tasks with significance less than 1.0, the programmer may provide an alternative, approximate

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

Listing 2: #pragma omp task

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.

Task input and output is explicitly specified via the *in()* and *out()* clauses. This information is exploited by the runtime to detect task dependencies. 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). A *taskwait* can serve as a global barrier, instructing the runtime to wait for all tasks spawned up to that point in the code. Alternatively, it can implement a barrier for a specific task group, if the *label()* clause is present; in this case the runtime system waits for the termination of all tasks of that group.

Importantly, *omp taskwait* can be used to control the minimum quality of application results. Using the *ratio()* clause, the programmer can instruct the runtime to execute (at least) the specified percentage of tasks – either globally or in a specific group, depending on the presence of a *label()* clause – in their accurate version, while *respecting* task significance (a more significant task should not be executed approximately, while a less significant task is executed accurately). The ratio 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. The runtime needs to ensure that at a minimum, the most significant 35% of the group tasks are executed accurately. Note that the runtime may opt for a higher ratio, provided this is feasible with the energy budget of the program.

3 Modeling and Prediction of Application Energy Footprint

We propose 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 and the mix of accurate and approximate tasks. We assume a general purpose shared-memory architecture with multiple multicore processors. All cores within each processor share the same last level cache. The model is trained through an offline profiling and fitting phase, discussed in Section 3.2.

3.1 Analytical model

3.1.1 Execution Time Modeling

Equation 1 models the sequential execution time of an application with m task-groups. Each task group i consists of n_i tasks, is executed with a ratio r_i and its accurate and approximate task functions execute in $\overline{T_{acc}}$ and $\overline{T_{app}}$ seconds on average, respectively. The average execution time of tasks depends on both problem size and the number of tasks; a larger problem size may require additional tasks to be created, or more work per task, or a combination thereof. Moreover, this allows the profile-driven model fitting phase to account for locality, caching and memory traffic effects due to different input and intermediate data footprints related to different problem sizes. These effects can significantly affect task execution time.

$$T_{seq}(\vec{r}, s) = \sum_{i=1}^{m} n_i \cdot (r_i \cdot \overline{T_{acc_i}(s, n_i)} + (1 - r_i) \cdot \overline{T_{app_i}(s, n_i)}) \tag{1}$$

$$T_{par}(\vec{r}, s, C) = \frac{T_{seq}(\vec{r}, s)}{C \cdot efficiency(s, C)} \tag{2}$$

Equation 2 estimates parallel execution time of the target application when executing on C cores. The $efficiency(s, C)$ term approximates application scalability on different numbers of cores. It depends on input problem size (s) and the number of cores (C). On a multiprocessor with multicore processors, we assume a packed thread allocation strategy, where all cores in a processor must be allocated to threads before allocating threads on another processor. This is the most energy efficient allocation strategy in the common case.

3.1.2 Power Modeling

The power consumption of processing units during program execution has two components, given in Equation 3.

$$P = P_{background}(C) + P_{dynamic}(C, s, \vec{r}) \tag{3}$$

$P_{background}$ corresponds to the "background" power consumed by active CPU cores (C), whereas $P_{dynamic}$ corresponds to the power component which varies depending on the computations executed on each core. The latter depends on the number of cores, application input size and application mix of accurate/approximate tasks.

3.2 Offline Profiling and Model Fitting

During profiling, each application is executed with three different, representative data-sets, of varying sizes ($size(s)$) and thus 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 application for all possible ($\vec{r}$,C) configurations. We measure the average execution time of approximate and accurate tasks,($\overline{T_{acc}}$ and $\overline{T_{app}}$) for each task group and the total execution time of each group. Finally, we measure the overall execution time and power consumption of the application.

Profiling is followed by a model fitting phase, during which profiling data serve as input to a regression process. The fitting phase predicts execution time and energy consumption of the application for different configurations in terms of problem size, number of cores and ratio of accurate to total number of tasks.

The first step of the fitting phase produces an estimation function of the average execution time of accurate and approximate tasks. We perform regression to map the average execution time of each

[1]We skip the large data-set for the Monte Carlo PDE solver because it produces unrealistic execution times.

category of tasks (approximate, accurate) to the problem size and number of tasks. A separate function is created for each possible core count. We use the average execution time of tasks observed when executing across all ratios. Exponential, polynomial and linear fitting functions are all attempted and we keep the one which achieves the most accurate estimations compared to profiling data. For the applications considered in this paper we note that the most frequently used function is of the form $Time_{task}(s, n_i) = a \cdot (\frac{s}{n_i})^b$, where a and b are the outputs of multivariate regression.

Next, we evaluate the function that computes the *efficiency* term, 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 core count, correlating efficiency to problem size.

Finally we evaluate the function that computes power (P). Profiling results indicated that on our specific experimental platform, (Intel Sandy Bridge, dual-processor, octa-core processors) $P_{dynamic}$ is negligible, whereas $P_{background}$ consists of two terms: one which is linearly dependent on the number of active CPU packages (sockets) and a second which is linearly dependent on the number of active cores. This observation was confirmed by our results of the fitting process using regression. Therefore, for our experimental platform, Equation 3 degenerates to a look-up table, connecting power to the number of cores used. The table is common for all applications, problem sizes and approximate to accurate task ratios.

4 RunTime System

We extend a task-based parallel runtime system that implements OpenMP 4.0-style task dependencies [17] 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 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 [18].

5 Experimental Evaluation

We use six benchmarks to validate our framework and its ability to execute applications with a predefined 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 [13] 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 [14]. 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 [19] 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 netlist 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 netlist. 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.

5.2 Experimental Methodology

The experimental analysis was carried out on a system equipped with two Intel(R) Xeon(R) E5-2650 processors clo-cked at 2.00 GHz, and 64 GB shared DRAM. Each processor has 8 cores. Energy and power are measured using likwid [16], which provides access to 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 configurations. From those executions we identify the one that consumes the least energy. If forfeiting quality is not an option for the end-user, this configuration represents the minimum energy consumption that can be achieved. 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 Results

Figure 1 summarizes our results. We present three charts for each benchmark, organized vertically and depicting Quality, Energy and Ratio/Cores. For the media applications *(DCT, Sobel, Fisheye)* the quality is quantified using the *PSNR* quality metric (higher is better). For *Kmeans*, the quality of output data is characterized by computing the average distance between data points and the center of the cluster they are assigned to (lower is better). Finally, for *MC* and *Canneal* we report the relative error with respect to an accurate execution (lower is better). In all charts the horizontal axis represents the requested energy budget, as a percentage of the energy consumed by the least energy consuming accurate execution. The vertical axis of the energy charts presents the energy that was actually consumed by approximate executions as a percentage of the energy consumed by the accurate execution. In quality and energy charts we present the quality/energy achieved by our framework, the optimal oracular executions (determined experimentally offline) and loop perforation. Finally the Ratio/Cores charts present the decisions of the runtime: the number of cores corresponds to bars, while the approximation degree is presented as lines. We compare the choices of our runtime to the optimal oracular configuration.

Our framework produces program configurations that always match or are very similar to the optimal ones. Even in the cases 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 and Kmeans which are discussed further later in this section. Both our approach and the optimal tend to adapt concurrency to utilize all cores of one or two CPUs, depending on the parallel efficiency of each application and the input size. This is expected, as the dominant term in power estimation is due to the activation of additional CPUs.

We observe that multimedia applications are well-suited for our programming framework. All three applications can execute with as little as 50% of the energy required by the optimum accurate execution and match the optimal achievable quality. The lowest achievable energy mainly depends on the complexity of the approximation function we use. At the same time the complexity of the approximation function dictates the quality of the output for the most aggressive degrees of approximation. When approximating all tasks we observe PSNR equal to 18.70, 28.99, 36.01 dB for *Sobel*, *DCT*, and *Fisheye* respectively. The

perforated execution, when consuming the same amount of energy produces inferior quality corresponding to PSNRs of 10.75, 13.75, 7.73 dB respectively. Our methodology clearly produces higher quality of results with the same energy budget. Multimedia benchmarks take full advantage of the significance and approximation features of the programming model. Moreover, they scale to larger inputs by adapting the number of tasks accordingly, therefore our model can predict their performance with high accuracy. Finally, the execution of approximate tasks has straightforward and easy to model effects on execution time: more approximate tasks result to less computation and thus energy savings.

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 to 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.

Our model is less accurate in its predictions for *Canneal*, compared to multimedia benchmarks. This is a consequence of the bad, unpredictable locality pattern of this application. Canneal uses large data structures to store information on netlist 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. This is particularly true for very tight energy budgets. That said, in all cases we achieve consistently better quality of results compared to perforation. It is also worth noting that the runtime consistently selects few cores to execute the application, which is expected due to the low efficiency with an increasing number of cores.

Kmeans shows the limitations of our approach. It is notoriously difficult to be modeled, 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). Moreover, even when we approximate 100% of the point classification tasks, we can only reduce the energy footprint by at most 50%. This is 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. It can reduce energy consumption to 20% of the baseline, keeping the relative error within 5%.

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.

Compared to approximate programming APIs for example Green [1] and stochastic processor software APIs [15], our programming model offers the higher level abstraction of computational significance for managing quality, while delegating the selection of tasks to approximate and the control of result quality to a dynamic optimizer and the runtime system. Contrary to approximation techniques via loop perforation implemented in the compiler [13], our programming model uses domain expertise available from the programmer, which we demonstrate to be necessary for effective approximation in at least one application domain. On the other hand, our programming model remains general-purpose, contrary to application-specific approximate code generators such as SAGE [10], Paraprox [9] ApproxIt [20] and related work on iterative solvers [4] and video codecs [12]. Compared to other approximate language frameworks such as EnerJ [11], our programming model offers additional features including task-parallel execution and energy-constrained runtime optimization of output quality.

Compared to implicitly parallel and explicitly parallel approximate computing frameworks, such as Quickstep [5], variability-aware OpenMP [8] and variation tolerant Open-MP [7], our programming and runtime system provides better optimization capability via selective execution of approximate code and dynamic control of accuracy and efficiency. We achieve this by providing concurrency control (elasticity) and the ability to execute in a range of operating points that meet quality and energy constraints.

7 Conclusion

This paper introduced the first 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 and the ratio of accurate versus approximate tasks. This model is exploited by the runtime system of an energy-constrained multicore platform to steer execution towards a configuration that maximizes quality of output while complying with energy constraints.

The experimental evaluation across several benchmark co-des 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 [13], 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 widen 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.

8 Acknowledgments

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] 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.

[6] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A. Skodras, C. Christopoulos, and T. Ebrahimi. The jpeg 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36–58, Sept. 2001.

[15] 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.

[16] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, Sept. 2010.

[17] 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.

[18] 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.

[19] M. Vavalis and G. Sarailidis. Hybrid-numerical-PDE-solvers: Hybrid Elliptic PDE Solvers. `http://dx.doi.org/10.5281/zenodo.11691`, Sep 2014.

[20] 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.

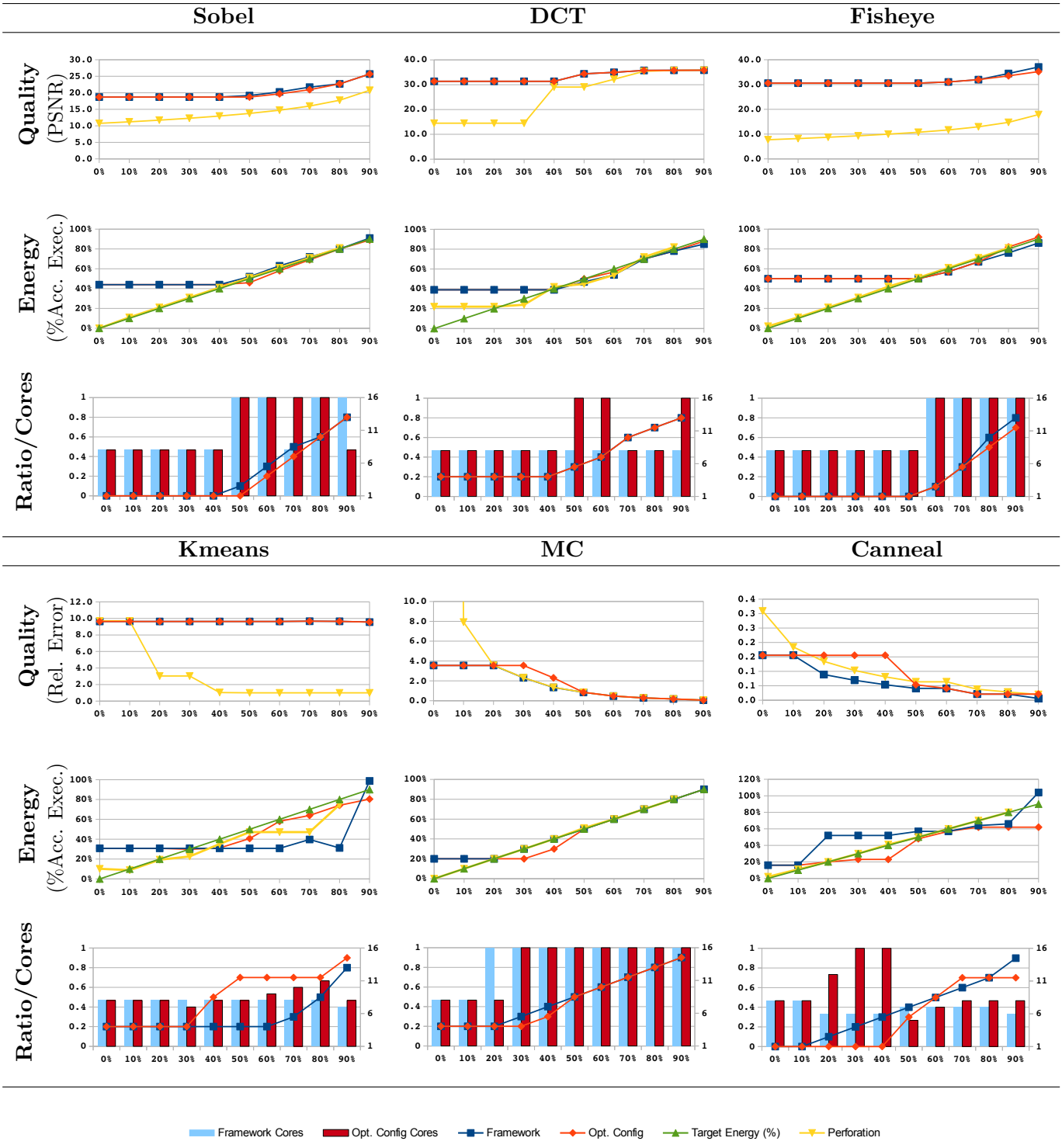

Figure 1: Quality, energy, and execution configurations for different energy targets (as a percentage of the most energy-efficient accurate execution). Energy & quality plots show the results achieved by our system, an oracle selecting the optimal configuration and loop perforation. The energy plots include a 45° line to visualize the ideal relation between achieved energy ratio (vertical axis), against the requested (horizontal axis). The configuration of the system compares the selected cores and approximation degree of the optimal solution with the respective ones selected by our framework.