A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing

Vassilis Vassiliadis*, Konstantinos Parasyris*, Charalambos Chalios†, Christos D. Antonopoulos*,
Spyros Lalis*, Nikolaos Bellas*, Hans Vandierendonck† and Dimitrios S. Nikolopoulos†
*University of Thessaly & CERTH, Greece
Email: {vasiliad, koparasy, cda, lalis, nbellas}@inf.uth.gr
†Queen's University of Belfast, UK
Email: {cchalios01, h.vandierendonck, d.nikolopoulos}@qub.ac.uk

***Abstract*—One factor that contributes to high energy consumption is that all parts of the program are considered equally significant for the accuracy of the end-result. However, in many cases, parts of computations can be performed in an approximate way, or even dropped, without affecting the quality of the final output to a significant degree. In this paper, we introduce a task-based programming model and runtime system that exploit this observation to trade off the quality of program outputs for increased energy-efficiency. This is done in a structured and flexible way, allowing for easy exploration of different execution points in the quality/energy space, without code modifications and without adversely affecting application performance. The programmer specifies the significance of tasks, and optionally provides approximations for them. Moreover, she provides hints to the runtime on the percentage of tasks that should be executed accurately in order to reach the target quality of results. Two different significance-aware runtime policies are proposed for deciding whether a task will be executed in its accurate or approximate version. These policies differ in terms of their runtime overhead but also the degree to which they manage to execute tasks according to the programmer's specification. The results from experiments performed using six different bencmharks on top of an Intel-based multicore/multiprocessor platform show that, depending on the runtime policy used, our system can achieve an energy reduction of up to 83% vs. a fully accurate execution, and up to 35% vs. approximate versions that employ loop perforation. At the same time, our approach always results to graceful degradation of the final result.**

I. Introduction

One factor that contributes to the energy footprint of current computer technology is that all parts of the program are considered to be equally important, and thus are all executed with full accuracy. However, as shown by previous work on approximate computing, in several classes of computations, not all parts or execution phases of a program affect the quality of its output equivalently. In fact, the output may remain virtually unaffected even if some computations produce incorrect results or fail completely.

In this paper, we introduce a novel, significance-driven programming environment for approximate computing, comprising a programming model, compilation toolchain and runtime system. The environment allows programmers to trade-off the quality of program outputs for increased energy-efficiency, in a structured and flexible way. The programming model follows a task-based approach. For each task, the developer declares its significance depending on how strongly the task contributes to the quality of the final program output, and provides an approximate version of lower complexity that returns a less accurate result or just a meaningful default value. The developer controls the degradation of output quality by specifying the percentage of tasks to be executed accurately. In turn, the runtime system executes tasks in a significance-aware fashion, by employing the approximate versions of less-significant tasks, or dropping such tasks altogether. This can lead to shorter makespans and thus to more energy-efficient executions, without having a significant impact on the final output.

Section II introduces the programming model. Section III discusses the runtime system, and the different policies used to drive task execution. Section IV presents the experimental evaluation on top of an Intel-based multiprocessor/multicore platform, using a set of benchmark kernels that were ported to our programming model. Section V gives an overview of related work. Finally, Section VI concludes the paper and identifies directions for future work.

II. Programming Model

Our vision is to elevate significance characterization as a first class concern in software development, similar to parallelism and other algorithmic properties traditionally being in the focus of programmers. To this end, the main objectives of our programming model are: (i) to allow programmers to express the significance of computations in terms of their contribution to the quality of the end-result; (ii) to allow programmers to specify approximate alternatives for selected computations; (iii) to allow programmers to express parallelism, beyond significance; (iv) to allow programmers to control the balance between energy consumption and the quality of the end-result, without sacrificing performance; (v) to enable optimization and exploration of trade-offs at execution time; and (vi) to be user-friendly and architecture neutral.

Programmers express significance semantics using *#pragma* compiler directives. Pragmas-based programming models facilitate non-invasive and progressive code transformations, without requiring a complete code rewrite. We adopt a task-based paradigm, similarly to the latest version of OpenMP [1]. Task-based models offer a straightforward way to express communication across tasks, by explicitly defining inter-task data dependencies. Parallelism is expressed by the programmer in the form of independent tasks, however the scheduling of the tasks is not explicitly controlled by the programmer, but is performed at runtime, also taking into account the data dependencies among tasks.

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

Listing 1: #pragma omp task

Tasks are specified using the *#pragma omp task* directive (Listing 1), followed by a function which is equivalent to the task body. The significance of the task is specified through the *significant()* clause. Significance takes values in the range [0.0, 1.0] and characterizes the relative importance of tasks for the quality of the end-result of the application. Depending on their (relative) significance, tasks may be approximated or dropped at runtime. The special values 1.0 and 0.0 are used for tasks that must *unconditionally* be executed accurately and approximately, 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 for a non-accurate computation of the task. It typically implements a simpler, approximate version of the computation, which may even degenerate to just setting default values to the output. If a task is selected by the runtime to be executed approximately, and the programmer has not supplied an *approxfun* version, the task is simply dropped. It should be noted that the *approxfun* function implicitly takes the same arguments as the function implementing the accurate version of the task body.

Programmers explicitly specify data flow to the task through the *in()* and *out()* clauses. This information is exploited by the runtime to automatically determine the dependencies among tasks. Finally, *label()* can be used to group tasks, and to assign the group a common identifier (name), which is in turn used as a reference to implement synchronization at the granularity of task groups (see next).

The proposed programming model supports explicit barrier-type synchronization through the *#pragma omp taskwait* directive (Listing 2). This 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 at the granularity of a specific task group, if the *label()* clause is present; in this case the runtime waits for the termination of all tasks of that group. Finally, the *on()* clause instructs the runtime to wait for all tasks that affect a specific variable or data construct.

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

Listing 2: #pragma omp taskwait

Furthermore, the *omp taskwait* barrier can be used to control the minimum quality of application results. Through the *ratio()* clause, the programmer can instruct the runtime to execute (at least) the specified percentage of all tasks – either globally or in a specific group, depending on the existence of the *label()* clause – in their accurate version, while *respecting* task significance (i.e., 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, however at a potential penalty on the quality of the produced output.

The compiler for the programming model is implemented based on a source-to-source compiler infrastructure [2]. It recognizes the pragmas introduced by the programmer and lowers them to corresponding calls of the runtime system discussed in the next section.

III. Runtime

To support the above programming mode, we have extended a task-based parallel runtime system that implements OpenMP 4.0-style task dependencies [3]. The runtime system is organized as a master/slave work-sharing scheduler. The master thread starts executing the main program sequentially. For every task call encountered, the task is enqueued in a per-worker task queue. Tasks are distributed across workers in round-robin fashion. Workers select the oldest tasks from their queues for execution. When the worker's own queue runs empty, the worker may steal one or more tasks from the queues of other workers.

A. API Extension

The runtime exposes an API that matches with the pragma-based programming model. Every pragma is translated in one or more runtime calls. The runtime API is extended to support the programming model as follows.

The *tpc_task_create()* primitive is extended to indicate the *task group* and *significance* of the task, as well as to supply an alternative (approximate) task function. Also, a new *tpc_init_group()* primitive is added, which is used by the compiler to pass information about a task group when its name is encountered for the first time in the program code. This call is used to create support data structures in

the runtime for the task group, as well as to convey the per-group ratio of tasks that must be executed accurately. Finally, next to the existing *tpc_wait_all()* call that waits for all tasks to finish, the new *tpc_wait_group()* primitive is introduced in order to wait for a specific task group to complete.

B. Task Execution

The runtime system has to selectively execute a subset of the tasks approximately, while respecting the ratio and task significance specified by the programmer. In general, the runtime system has no a priori information on how many tasks will be issued in a task group, nor what the distribution of significance levels in each task group will be. This information must be collected at runtime. For this purpose we have designed and implemented two task selection policies, briefly described below.

1) Global Task Buffering (GTB): In this policy the master thread buffers tasks as it creates them, postponing the issue of tasks in the worker queues. When the buffer is full, or when a call to *tpc_wait_all()* or *tpc_wait_group()* is made, the tasks in the buffer are sorted by significance, and the most significant ones are selected for accurate execution according to specified ratio (the rest are executed approximately). If the buffer size is sufficiently large, the runtime will buffer all tasks until the corresponding synchronization barrier is encountered, and thus take a fully correct decision as to which tasks to run accurately/approximately. However, a large buffer size slows down execution, as the runtime postpones tasks until the buffer is filled. This problem can be mitigated by using a smaller buffer window size and tasks of coarse enough granularity, so that the runtime system can overlap task issue with task execution. In our implementation, the buffer size is a configurable parameter passed to the runtime system at compile time.

2) Local Queue History (LQH): The local queue history policy avoids task buffering at the master. Tasks are issued to worker queues immediately as they are created. The worker decides whether to approximate a task before starting its execution, based on the distribution of significance levels of the tasks executed so far, trying to converge to the ratio specified by the programmer. The overhead of the local queue history algorithm is the bookkeeping of the statistics on the execution history of a group. This happens every time a task is executed. Updating statistics includes accessing an array of size equal to the number of distinct significance levels, which is negligible compared to the granularity of the task. Importantly, in LQH, each worker takes decisions in a fully distributed way, using only local information from the tasks that appear in its work queue, without any coordination. It is thus more realistic and scalable than GTB. However, given that each worker has only a local view of the tasks issued, it may fail to meet the quality requirements set by the programmer.

Benchmark	**A**pproximate or **D**rop	Approx Degree			Quality
		Mild	Med	Aggr	
Sobel	A	0%	30%	80%	PSNR
DCT	D	10%	40%	80%	PSNR
MC	D, A	50%	80%	100%	Rel. Err.
Kmeans	A	40%	60%	80%	Rel. Err.
Jacobi	D, A	10^{-4}	10^{-3}	10^{-2}	Rel. Err.
Fluidanimate	A	12.5%	25%	50%	Rel. Err.

Table I: Benchmarks used for the evaluation. For all cases, except Jacobi, the degree of approximation is given by the percentage of tasks executed approximately. In Jacobi, it is given by the error tolerance in convergence of the accurately executed iterations/tasks (10^{-5} in the native version).

IV. Experimental Evaluation

We performed a set of experiments to investigate the performance and energy-reduction potential of the proposed programming model and runtime policies. In the sequel, we introduce the benchmarks used, the overall evaluation approach, and discuss the results achieved for various degrees of approximation under different runtime policies.

A. Benchmarks and Approach

We use six different benchmark codes, which were re-written using our task-based pragma directives: (i) the *Sobel* filter; (ii) the Discrete Cosine Transform (*DCT*), used in JPEG compression and decompression; (iii) *MC* [4] which applies a Monte Carlo approach to estimate the boundary of a subdomain within a larger partial differential equation (PDE) domain by performing random walks; (iv) *K-means* clustering; (v) the *Jacobi* iterative solver for diagonally dominant systems of linear equations; and (vi) *Fluidanimate*, a code from the PARSEC suite, which applies the smoothed particle hydrodynamics (SPH) method to compute the movement of a fluid in consecutive time steps.

For each benchmark we apply different approximation approaches and/or drop tasks, subject to the characteristics of the respective computation (the details are omitted due to space limitations). Also, three different degrees of approximation are studied for each benchmark: *Mild, Medium,* and *Aggressive* (see Table I). They correspond to different choices in the quality vs. energy and performance space. It should be noted that, with the partial exception of *Jacobi*, quality control is possible solely by changing the *ratio* parameter of the *taskwait* pragma. This is indicative of the flexibility of our programming model.

The quality of the result is evaluated by comparing it to the output produced by a fully accurate execution. The quality metric depends on the computation. For the image processing benchmarks *DCT* and Sobel we use the peak signal to noise ratio (*PSNR*), whereas for *MC, Kmeans, Jacobi* and *Fluidanimate* we use the relative error.

In the experiments, we measure the performance of our approach for each benchmark and approximation degree,

using both runtime policies GTB and LQH. Also, for GTB, we investigate two different cases: using a sufficiently large buffer so that all tasks of a group are buffered until the synchronization barrier, referred to as *Max-Buffer GTB*; using a smaller buffer depending on the computation, so that task execution can start earlier, called *User-Defined GTB*.

As a reference, we compare our approach against: (i) a fully accurate execution on top of a significance-agnostic version of the runtime system, and (ii) an execution using loop perforation [5], a simple yet usually effective compiler technique for approximation. Loop perforation is also applied in three different degrees of aggressiveness. The perforated version executes the same number of tasks as those executed accurately by our approach.

We run our experiments on a system equipped with 2 *Intel(R) Xeon(R) E5-2650* CPUs clocked at 2.00 GHz, with 64 GB shared memory. Each CPU consists of 8 cores (cores support hyper-threading, but we deactivated this feature in our experiments). We use Centos 6.5 Linux Operating system with the 2.6.32 Linux kernel. Each execution pinned 16 threads on all 16 cores. The energy and power are measured using likwid [6] to access the Running Average Power Limit (RAPL) registers of the processors.

B. Experimental Results

Figure 1 depicts the results of our experiments. For each benchmark we present execution time, energy consumption and the corresponding error metric. As can be seen, the approximated benchmark versions execute significantly faster and with less energy consumption compared to the accurate versions. Although output quality deteriorates as the approximation level increases, this typically occurs in a graceful way.

Overall, the two GTB policies exhibit similar performance. Even though Max-Buffer GTB postpones task issue until the creation of all tasks in the group, this does not seem to penalize execution. The reason is that in most benchmarks tasks are coarse-grained and organized in relatively small groups, thereby minimizing the task creation overhead and the latency for the creation of all tasks within a group. LQH is typically faster and more energy-efficient than GTB, except for *Kmeans* (this will be discussed in the sequel).

In *Sobel* the perforated version seems to significantly outperform our approach in terms of both energy consumption and execution time. However this is done at the cost of unacceptable output quality, even for the mild approximation level. Our programming model and runtime policies achieve graceful quality degradation, resulting to acceptable output even with aggressive approximation.

DCT is friendly to approximations: it produces acceptable results even if a large percentage of the computations is dropped. Our policies, with the exception of Max-Buffer GTB, perform comparably to loop perforation in terms of performance and energy consumption, but produce result of higher quality (note that PSNR is a logarithmic metric). This is due to the fact that our model offers more flexibility than perforation in defining the relative significance of code regions in DCT. The problematic behavior of Max-Buffer GTB is due to the fact that DCT creates many lightweight tasks. Given that in this case task creation is a non-negligible percentage of the total execution time, the latency between task creation and task issue imposed by the policy results to a notable overhead; especially in the case of aggressive approximation, where the ultimate decision will be to drop a large number of the created tasks anyway.

The approximate version of *MC* significantly outperforms the accurate version, without a big penalty on output quality. Randomized algorithms are inherently susceptible to approximations without requiring much sophistication. Thus the performance of our approach is almost identical to that of blind loop perforation. The LQH policy is faster and consumes less energy than GTB. However, in the aggressive and medium approximation degrees, LQH selects for approximate execution 4.6% and respectively 5.1% more tasks than it should vs. the programmer-specified ratio, which in turn affects output quality.

Kmeans behaves gracefully as the degree of approximation increases. Even in the aggressive case, all policies demonstrate relative errors less than 0.45%. The GTB policies are superior in terms of execution time and energy compared to the perforated version of the benchmark. LQH performs notably worse. This is beceause our termination criterion takes into account only object movements that have been computed accurately, However, in LQH, task approximation decisions are taken by each worker independently, so the set of objects that is computed accurately can vary in each iteration, which affects convergence.

Jacobi is a special case, in the sense that approximation can affect its rate of convergence in a way that is deterministic but nevertheless hard to predict and analyze. The blind perforation version requires fewer iterations to converge, thus resulting to lower energy consumption than our policies. Interestingly enough, it also results to a solution closer to the real one, compared with the accurate execution.

Perforation could not be applied in *Fluidanimate*. If the movement of some particles during a time-step is skipped, the physics of the fluid are violated, leading to completely wrong results. Moreover, to ensure stability it is necessary to alternate between fully accurate and approximate time steps. Our programming model allows to approximate the computation in a controlled way, by alternatively changing the *ratio* at the *taskbarrier* between consecutive time steps, from 1.0 to a smaller value (so that tasks can run approximately). Note that due to the sensitivity of *Fluidanimate* to errors, aggressive approximation with LQH leads to unacceptably bad results. However, at a medium degree of approximation, LQH achieves good results at less than half the energy of the accurate execution, with GTB being almost as efficient.

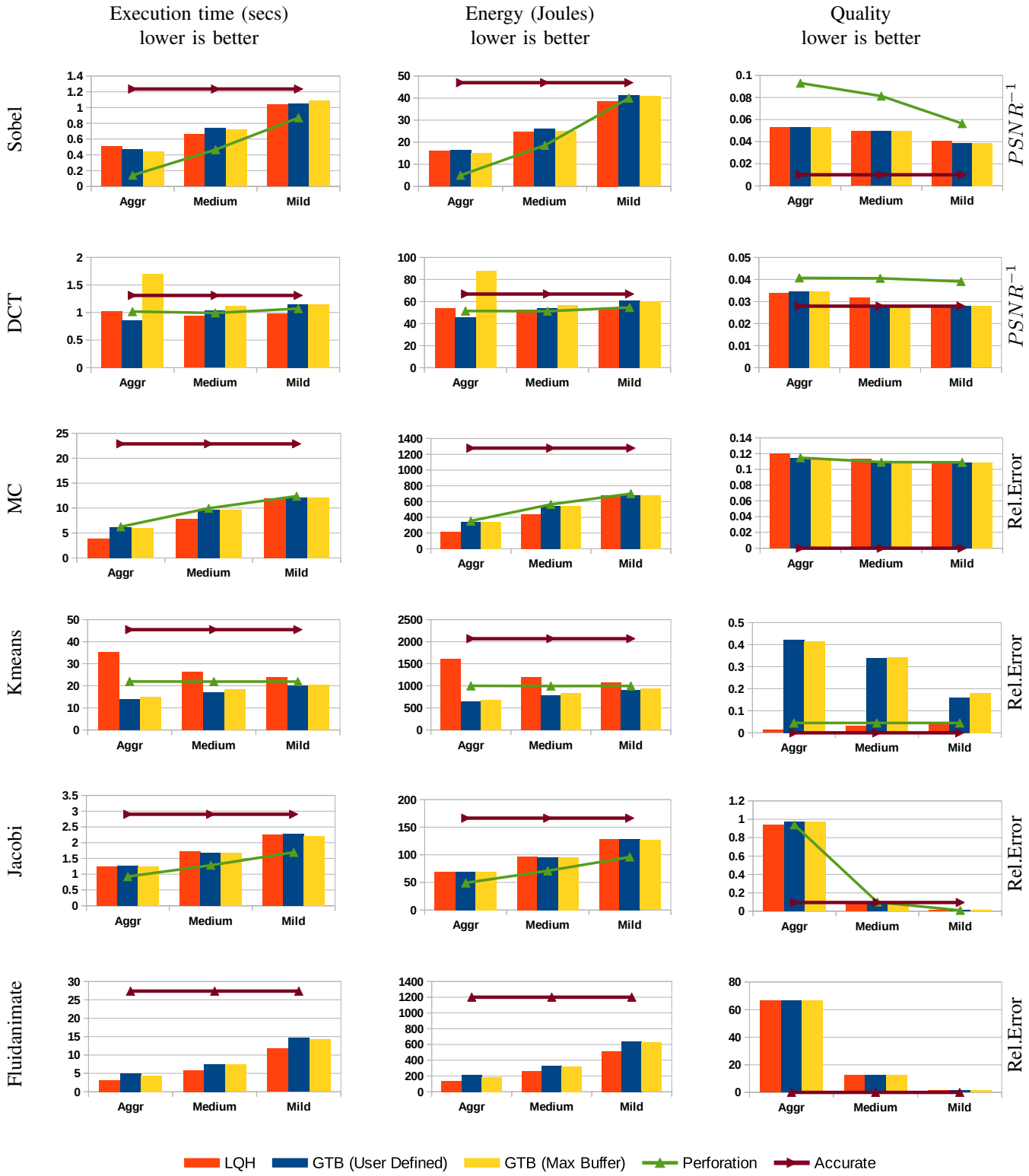

Figure 1: Execution time, energy and quality of results for the benchmarks used in the experimental evaluation under different runtime policies and degrees of approximation. In all cases lower is better. Quality is depicted as PSNR^{-1} for Sobel and DCT, relative error (%) is used in all others benchmarks. The accurate execution and the approximate execution using perforation are visualized as lines. Note that perforation was not applicable for Fluidanimate.

V. Related Work

Green [7] 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 code blocks, as well as re-calibration functions for correcting unacceptable errors due to approximation. Loop perforation [5] is a compiler technique that classifies loop iterations into critical and non-critical ones. The latter can be dropped, as long as the results are acceptable. In our approach, such optimizations are driven by the relative significance of code blocks, and are applied selectively at runtime to meet user-defined quality criteria.

EnerJ [8] implements approximate data types and supports user-defined "approximable" methods, without tying these abstractions to a specific approximate execution model. Similarly to our framework, EnerJ provides abstractions that allow the programmer to provide hints on where approximate execution can be safely used in a program and the prototype version runs on a simulated environment. Contrary to our framework, EnerJ does not use a runtime substrate for approximation on general-purpose hardware and does not consider code dropping or task-parallel execution.

Variability-aware OpenMP [9] is a set 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. However, approximation applies only to FPU operations, which execute on special FPUs with configurable accuracy. Our framework applies selective approximation at the granularity of tasks, using the significance abstraction, thereby providing the flexibility to drop or approximate code while preserving output quality. Furthermore, our framework does not require specialized hardware support.

ApproxIt [10] 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. Schmoll et al. [11] 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.

ERSA [12] is a multi-core architecture where cores are either fully reliable or have relaxed reliability. The program is divided into critical (typically control code) and non-critical (typically data processing code) parts, which are assigned to reliable or unreliable cores, respectively. Thus ERSA uses an explicit and application-specific assignment of code to cores with different levels of reliability. We follow a different approach whereby the programmer uses significance to implicitly indicate code that can be approximated, and the runtime system implements selective approximation. In our framework, accurate and approximate code may run on any core for load balancing purposes.

VI. Conclusions

We introduced a programming model that supports approximate computing at the granularity of tasks, and developed corresponding runtime support for elastically deciding which tasks to execute approximately or drop completely, while meeting the specified quality/accuracy target.

In the future, we wish to explore more optimization scenarios, such as DFVS in conjunction with suitable runtime policies for executing approximate (and more light-weight) task versions on the slower but also less power-hungry CPUs, as well as for using more such cores to make up for this slower execution. We are also interested in extending our programming model to support approximate computing on top of ultra low-power but unreliable hardware.

References

[1] O. A. R. Board, "OpenMP Application Program Interface (version 4.0)," Tech. Rep., Jul. 2013.

[2] F. S. Zakkak, D. Chasapis, P. Pratikakis, A. Bilas, D. S. Nikolopoulos, "Inference and declaration of independence in task-parallel programs", *Advanced Parallel Processing Technologies*, LNCS 8299, pp. 1–16, 2013.

[3] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, D. S. Nikolopoulos, "Bddt: Block-level dynamic dependence analysis for deterministic task-based parallelism," *SIGPLAN Not.*, 47(8), pp. 301–302, Aug. 2012.

[4] M. Vavalis, G. Sarailidis, "Hybrid-numerical-pde-solvers: Hybrid elliptic pde solvers," Sep. 2014. [Online]. Available: http://dx.doi.org/10.5281/zenodo.11691

[5] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," *ACM SIGSOFT Symposium and ESEC/FSE*. pp. 124–134, 2011.

[6] J. Treibig, G. Hager, G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," *ICPP Workshops*, pp. 207–216, 2010.

[7] W. Baek. T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," *PLDI*, pp. 198–209, 2010.

[8] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," *PLDI*, pp. 164–174, 2011.

[9] A. Rahimi, A. Marongiu, R. K. Gupta, L. Benini, "A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters," *CODES+ISSS*, pp. 35:1–35:10, 2013.

[10] Q. Zhang, F. Yuan, R. Ye, Q. Xu, "Approxit: An approximate computing framework for iterative methods," *DAC*, pp. 97:1–97:6, 2014.

[11] F. Schmoll, A. Heinig, P. Marwedel, M. Engel, "Improving the fault resilience of an h.264 decoder using static analysis methods," *ACM TECS*, 13(1s), pp. 31:1–31:27, Dec. 2013.

[12] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *DATE*, pp. 1560–1565, 2010.