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

Vassilis Vassiliadis[1] Konstantinos Parasyris[2] Charalambos Chalios[3] Christos D. Antonopoulos[4]
Spyros Lalis[5] Nikolaos Bellas[6] Hans Vandierendonck[7] Dimitrios S. Nikolopoulos[8]

[1,2,4,5,6]Electrical and Computer Eng. Dept. University of Thessaly, Greece [1,2,4,5,6]Centre for Research and Technology Hellas (CE.R.T.H.), Greece [3,7,8]Queen's University Belfast United Kingdom

{vasiliad[1],koparasy[2],cda[4],lalis[5],nbellas[6]}@uth.gr {cchalios01[3],h.vandierendonck[7],d.nikolopoulos[8]}@qub.ac.uk

Abstract

We introduce a task-based programming model and runtime system that exploit the observation that not all parts of a program are equally significant for the accuracy of the end-result, in order to trade off the quality of program outputs for increased energy-efficiency. This is done in a structured and flexible way, allowing for easy exploitation of different points in the quality/energy space, without adversely affecting application performance. The runtime system can apply a number of different policies to decide whether it will execute less-significant tasks accurately or approximately. The experimental evaluation indicates that our system can achieve an energy reduction of up to 83% compared with a fully accurate execution and up to 35% compared with an approximate version employing loop perforation. At the same time, our approach always results in graceful quality degradation.

Categories and Subject Descriptors D.3.4 [*Processors*]: Optimization

Keywords Energy saving, approximate computing, programming model, runtime system

1. 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 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. Also, 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 on available cores 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 results of the computation. The reader can find an extended version of this manuscript in [2].

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688546

2. Programming Model

The programming model allows the programmer to express her perspective on the significance of the contribution of each computation to the quality of the final output using compiler directives. Significance can be specified statically, or as an expression evaluated at run-time, during task creation. It characterizes the relative importance of tasks for the quality of the end-result of the application. The programmer may provide an alternative, approximate task body, which 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. Finally, tasks can be grouped and assigned a common name, which is used as a reference to implement synchronization at the granularity of task groups.

The programming model supports barriers at a global- or at a task group-level. Barriers can be used to control the minimum quality of application results. The programmer can instruct the runtime to execute (at least) the specified percentage (ratio) of all tasks – either globally or in a specific group – 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 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 quality penalty.

The compiler for the programming model recognizes the pragmas introduced by the programmer and lowers them to corresponding calls of the runtime system.

3. Runtime System

The runtime system selectively executes a subset of the tasks approximately while respecting the constraints given by the programmer. The relevant information consists of (i) the significance of each task, (ii) the group a task belongs to, and (iii) the fraction of tasks that may be executed approximately for each task group. Obviously, preference should be given to approximating tasks with lower significance values. The runtime system has no a priori information on how many tasks will be issued in a task group, nor on the distribution of the significance levels. This information must be col-

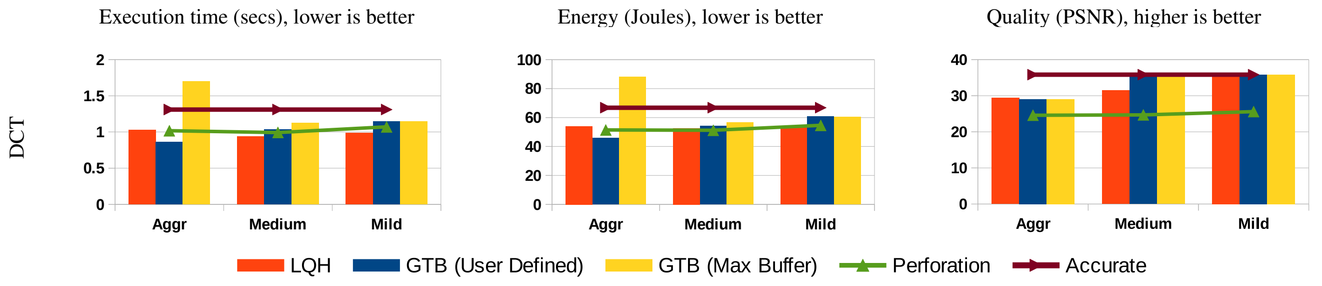

Figure 1: Execution time, energy and quality of results for DCT under different runtime policies and degrees of approximation. The accurate execution and the approximate execution using perforation are visualized as lines.

lected at runtime. We define two policies: one globally controlled, based on buffering issued tasks and analyzing their properties, and a policy that estimates the distribution of significance levels using per-worker local information.

In the Global Task Buffering (GTB) policy, spawned tasks are buffered, postponing their issue to the worker queues. Then, the tasks in the buffer are analyzed and sorted by significance. Given a per-group ratio of accurate tasks R_g, and a number of B tasks in the buffer, then the $R_g \cdot B$ tasks with the highest significance level are executed accurately. The tasks are subsequently issued to the worker queues. The task buffering policy is parameterized by the task buffer size. A larger buffer allows the runtime to take more informed decisions. However, at the same time it increases the latency between task spawning and execution.

The Local Queue History (LQH) policy avoids the step of task buffering. Tasks are issued to worker queues immediately as they are created. The worker decides whether to approximate a task right before it starts its execution, based on the distribution of significance levels of the tasks it has executed so far, and the target ratio of accurate tasks (supplied by the programmer). Hereto, the workers track the number of tasks at each significance level as they are executed. The runtime targets a ratio of accurately executed tasks that converges to the one specified by the programmer and approximates those tasks with the lowest significance level. The overhead of the local queue history algorithm for maintaining the statistics that form the execution history of a group is negligible. The policy requires no global information and synchronization. It is thus more realistic and scalable than GTB. However, given that each worker has only a localized view of the tasks issued, the runtime system can only approximately enforce the quality requirements set by the programmer.

4. Experimental Evaluation

We use a set of six benchmarks where we apply different approximation approaches, subject to the nature/characteristics of the respective computation. Due to space limitations Figure 1 presents the results for DCT only. Three different degrees of approximation are studied for each benchmark: *Mild, Medium,* and *Aggressive*. They correspond to different choices in the quality vs. energy and performance space. Quality control is possible solely by changing the *ratio* parameter of task group barriers. In the experiments, we measure the efficiency of our approach for the two different runtime policies GTB and LQH. For GTB, we investigate two cases: the buffer size is set so that tasks are buffered until the synchronization barrier (referred to as Max Buffer GTB); the buffer size is set to a smaller value, depending on the computation, so that task execution can start earlier (referred to as User Defined GTB). As a reference, we compare our approach against a fully accurate execution of each application, and an execution using loop perforation [1]. The experimental evaluation is carried out on a system equipped with 2 *Intel(R) Xeon(R) CPU E5-2650* processors clocked at 2.00 GHz, with 64 GB shared memory. Each CPU consists of 8 cores and all benchmark executions used 16 threads.

In *DCT* we assign higher significance to tasks that compute lower frequency coefficients, because human eye is more sensitive to low frequencies. As a quality metric, we use the PSNR of the produced image with respect to a “golden” image produced by the fully accurate execution. Note that PSNR is a logarithmic metric. DCT produces visually acceptable results even if a large percentage of the computations is dropped. Our policies, with the exception of the Max Buffer version of GTB, perform comparably to loop perforation in terms of performance and energy consumption, yet resulting in higher quality results. This is due to the fact that our model allows the programmer to define the relative significance of code computing different frequency coefficients. This information stems from algorithmic and physics properties of image processing and can not be attained by compiler analysis. Furthermore, the quality achieved by the educated approximation decisions taken by our model can not be achieved by blindly dropping computations, as is the case with loop perforation. The problematic performance of GTB (Max Buffer) is due to the fact that DCT task creation is a non-negligible percentage of the total execution time, therefore the latency between task creation and task issue introduced by the Max Buffer version of GTB results in a measurable overhead.

In general, across all applications, our system achieved an energy reduction of up to 83% compared with a fully accurate execution and up to 35% compared with an approximate version employing loop perforation. At the same time, our approach always resulted in graceful quality degradation.

Acknowledgements

This work has been partially supported by the European Commissions Seventh Framework Programme (FP7/2007- 2013) under grant agreement FP7-323872 (Project “SCoRPiO”) and under grant agreement FP7-327744 (NovoSoft, Marie Curie Actions).
This work has been partially supported by the “Aristeia II” action (grant agreement 5211, project “Centaurus”) of the operational program Education and Lifelong Learning and is co-funded by the European Social Fund and Greek national resources.

References

[1] 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.

[2] 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. Technical Report 2014-1, Department of Electrical and Computer Engineering, University of Thessaly, Greece, December 2014.