Approximate Computing: Facing The Control Flow

M. Ammar Ben Khadra
University of Kaiserslautern
khadra@eit.uni-kl.de

Dominik Stoffel
University of Kaiserslautern
stoffel@eit.uni-kl.de

Wolfgang Kunz
University of Kaiserslautern
kunz@eit.uni-kl.de

***Abstract*—We discuss Approximate Computing (AC) within a structured perspective where we propose a taxonomy and identify key challenges that need to be addressed in order for AC to be principled and automatic. To this end, we believe that AC programming support is a critical obstacle. Currently, AC-aware programming is mainly supported using general purpose languages, extended with annotations, that use local (loop-level) analyses. Instead, we argue that AC programming should emphasize parallelism, heterogeneity, and analyzability. The latter is particularly important for *global* (kernel-level) analyses to become effective. Finally, we will provide quantitative data for loop analyses of the PARSEC benchmark suite. We investigate properties related to *safe* approximation of loops.**

I. Introduction

Recent years have witnessed a surge in Approximate Computing (AC) research. This is motivated by the opportunities that emerging applications provide in terms of trading computational precision for energy savings and/or performance. These applications are collectively referred to as Recognition, Mining, and Synthesis (RMS) applications. Such AC-friendly applications can have (1) inputs drawn from the real world that can, therefore, be redundant or noisy, (2) computation patterns that employ iterative (self-healing) or probabilistic algorithms, or (3) outputs that are not exact or perceptually indistinguishable for end users. Basically, the main driving forces behind AC can be summarized in the following:

Big data. Over the next decade, the number of servers worldwide will grow by a factor of 10 while the amount of information managed by data centers will grow by a factor of 50. Therefore, throughput and storage need to be optimized.

Dark silicon. The end of Dennardian scaling has partially forced the move towards more cores per chips. However, scaling is still coupled with increase in power densities which limits the number of transistors that can be simultaneously powered on.

Reliability. Traditionally, software relied on a "guaranteed" service model provided by hardware. Such fixed guarantees are increasingly inefficient to maintain as variability and failure susceptibility continue to increase with more transistor scaling. Hence, exposing a "best-effort" service model to software is interesting to explore.

An ideal AC design flow begins with a program P, a quality specification QS, and a set of representative inputs $I_R \subseteq I_E$ where I_E represents expected inputs. I_R helps in identifying hotspots and expected run-time variable values. Later, an analysis A_{ac} should identify computations $C_p = \{C_p^i \,|\, i \in \mathbb{N}\}$ in P such that applying a set of transformations T_p to them would result in a program P' producing outputs O_p' that satisfy QS. Basically, an A_{ac} should do one or more of the following: (1) synthesize computations C_p' that approximate the original C_p in the domain of I_E, (2) synthesize optimized run-time configurations, e.g., for approximate memory allocation or arithmetic bit-precision, and (3) synthesize "application-aware" run-time quality management mechanisms.

Fine-grained (FG)	Coarse-grained (CG)
Fixed hardware (FH)	Configurable hardware (CH)
Guaranteed service (GS)	Best-effort service (BE)

Figure 1: AC taxonomy: an A_{ac} can either be FG or CG. A_{ac} can rely on FH or CH. Hardware can provide GS or BE. Right-side represents more efficiency opportunities

II. Challenges For AC

Manual (ad hoc) optimization of algorithms for a particular application and/or hardware architecture is already an established engineering practice. Hence, the AC value proposition lies in (1) automating the process of discovering quality, performance, and/or energy pareto-points satisfying QS, (2) providing "guarantees" that O_p' would always satisfy QS for I_E, and (3) making such a process applicable to a wide range of RMS applications. However, state-of-the-art AC research is still far from achieving such *ideal* goals due to many challenges. First, it is difficult to design a QS representation that is general enough to be applicable for the wide variety of RMS applications and is, yet, amenable to automatic analysis. For instance, quality of web search is based on "relevance" of results to given search terms, while quality of a clustering algorithm is measured (among others) with its inter-cluster "similarity". The task becomes even harder when QS involves human perception.

Second, AC research has been largely focusing on local function or loop-level optimizations which are, traditionally, also the focus of automatic parallelization research. The fundamental difference, however, is that it can be difficult to reason about the quality of end results E after applying a set of local T_p that introduce errors. This is particularly the case when computations C_p are not independent of each other, i.e. their results need to be composed to produce E. Third, there is the issue of maintaining QS guarantees for I_E given that A_{ac} only considers a subset $I_R \subset I_E$. Moreover, exposing hardware errors to software further complicates guaranteeing QS.

III. A Taxonomy for AC

We discuss in this section a taxonomy for AC methods. It is intended to create a better understanding of the various techniques proposed in the literature. Our taxonomy is highlighted in Fig. 1. First, an AC method may rely on a fixed hardware configuration (FH) or it may be able to adjust the hardware for its particular AC requirements, i.e., the hardware is configurable (CH). A software task can reconfigure the hardware

Table I: Classification of some AC works

	FH	CH
FG	Loop perforation [4]	Quora[5] Parrot [2]
CG	SAGE [3]	Parrot [2]

either at *load time* or dynamically at *run-time*. Second, an AC method is considered to be fine-grained (FG) if it focuses on loop-level analysis. On the other hand, it is considered coarse-grained (CG) if it focuses on whole computational kernels. Finally, the hardware might be restricted to providing a guaranteed service (GS) to software. Such service might be offered using instructions with a fixed error margin. Otherwise, the hardware can provide a best-effort service (BE) where the output of a computation, e.g., an instruction, or a load from storage is expected to be precise with a probability $p < 1$, while no assumptions can be made about the output in the remaining case. Note, however, that the hardware must provide a guaranteed service for computing control-flow tasks even under the best-effort model.

Some prominent AC works are classified according to our taxonomy in Table I. Note that our classification is not necessarily exclusive. For example, a software-only technique such as loop perforation [4] may also be utilized in CG-AC. Moreover, while the focus of the Parrot transformation to neural networks is largely FG-AC it is also applicable to CG-AC as long as its kernels are pure functions. Additionally, while more works have focused on hardware with a traditional guaranteed service, there are some notable works that considered the best-effort model, e.g., Esmaeilzadeh et al. [1].

IV. Facing the control flow

AC leverages the relaxed precision required to compute outputs. However, there is an unavoidable control overhead that needs to remain computed precisely even under a BE hardware model. Such control overhead needs to be minimized to better leverage efficiency opportunities. Examples of control overhead across the computing stack include:

Processors. Supplying instructions and pipeline management represent a significant control overhead. This overhead can be amortized over more data using SIMD coprocessors available in modern processors. However, the amount of data parallelism that a compiler can automatically extract is limited.

OSs. General-purpose OSs are essentially resource managers for competing processes. Consequently, they introduce control overhead that can reach as much as 33% . Therefore, there is a need for better static analyzability of resource utilization of each process. For example, processes arranged as synchronous data flow (SDF) can be statically scheduled.

Applications. Applications need to follow their control flow in order to produce outputs. However, the control flow might be dependent on output data. In this case, data approximation might be *unsafe* by introducing behaviors, e.g., exceptions, not available in the original program.

Better AC programming support is essential in order to tackle control overhead across the stack and address the challenges faced by AC (Sec. II). So far, AC-aware programming has been mainly based on local analyses for general purpose languages which are extended with annotations. Instead, we believe that programming for AC should emphasize parallelism (both at task and data levels), analyzability, and heterogeneity. Parallelism is essential in order to utilize the available hardware parallelism. We further elaborate on the latter criteria,

Static analyzability. Compiler optimizations are traditionally conservative in order to maintain software correctness. However, AC requires breaking such tradition by introducing "controlled" computation errors. Error controllability is challenging especially in the case of interdependent C_p. To cope with this, sensible restrictions on language features, e.g, pointer arithmetic, and models of computations are needed. For examples, adhering to a data flow model of computation explicitly exposes task composition to produce E. That enables A_{ac} to rely on QS of E more than local annotations. Also, it arranges tasks in a partial order which improves schedulability.

Embracing heterogeneity. The quest for better energy efficiency mandates moving towards *off-load* computing where performance critical code gets off-loaded to accelerators. While general purpose accelerators, e.g, GPGPUs and Xeon Phi, provide a good flexibility/efficiency design point, there is a need for more specialized accelerators. Such a need can be witnessed in mobile SoCs as well as in data centers, e.g., Google's TensorFlow Processing Unit. Moreover, research has demonstrated the value of on-chip, off-chip and even inmemory accelerators which will bring even more heterogeneity to the computing landscape. Hence, heterogeneous programming paradigms, e.g. OpenCL, need to be adopted for AC. In such paradigms, computations should be arranged in *kernels* that can run on specific *devices*.

V. Conclusion

AC is a fundamental tool in the quest for better efficiency. However, achieving *principled* and *automatic* AC will require us to compromise on generality across the computing stack. Basically, custom AC methods based on FH are already available in practice. Moving towards CH using algorithmic knobs in specialized accelerators is fruitful and within reach. Moving further towards BE hardware will remain challenging. However, specialized accelerators can provide a gradual path for BE hardware adoption where control-intensive tasks, e.g. OSs, remain mapped to reliable cores. Finally, our analysis results will focus on safe approximation of loops. We quantify properties like dependence of control on output data and usage of non-pure functions.

References

[1] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture support for disciplined approximate programming. In *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems* (2012), p. 301.

[2] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural Acceleration for General-Purpose Approximate Programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (dec 2012), pp. 449–460.

[3] Samadi, M., Lee, J., Jamshidi, D. A., Hormati, A., and Mahlke, S. SAGE: self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46* (2013), pp. 13–24.

[4] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. C. 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 - SIGSOFT/FSE '11* (2011), p. 124.

[5] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), pp. 1–12.