Overcoming the Data-flow Limit on Parallelism with Structural Approximation

Vignesh Balaji
Carnegie Mellon University
vigneshb@andrew.cmu.edu

Brandon Lucia
Carnegie Mellon University
blucia@andrew.cmu.edu

Radu Marculescu
Carnegie Mellon University
radum@andrew.cmu.edu

I. INTRODUCTION

Research in approximate computing is based on the observation that many important emerging applications tolerate errors. Their resilience allows system designers to deliberately execute programs imprecisely in a way that increases performance or decreases energy consumed. There are many examples of such error-performance tradeoffs in approximate computing literature [3], [1], [4], [9], [13], [10], [6].

The main idea driving most of these prior approximate computing efforts is that output *values* are approximable. These systems exploit that property through what we refer to as *value approximation*. Value approximation reduces the precision of [13], [10], [6], [12], or directly approximates [3], [1] a computation, producing approximate output values. Recent work extended the idea of value approximation to incorporate manipulations of algorithmic convergence criteria [9] and we refer to this extended approach as *convergence approximation*.

The unifying characteristic of all of these techniques is that they only approximate single values or collections of values for which a numerical comparison to a precise baseline is possible. The *error* induced by the approximation is a simple, numerical quantity. Approximating only simple, numeric quantities is a fundamental limitation of prior approaches to approximate computing.

It is our position that future computer systems should go beyond simply approximating values, also approximating the consistency properties of the data structures in a program. We refer to this approach as *Structural Approximation* (SA).

The core idea of SA is that applications can continue to execute even when their data structures are slightly "damaged" (i.e., inconsistent with their structural specification). We anticipate that through a combination of inherent application resiliance to damage, and surgically applied mitigations to damage, we can aggressively optimize data structure manipulations by making them approximate. However, to make our approximations *useful*, SA demands a new definition of *correctness* that allows systematically reasoning about the severity of structural inconsistency.

The motivation for SA is accelerating parallel computations that manipulate shared data structures. Parallel computations spend substantial time and energy to *enforce* structural consistency by moving data and synchronizing to serialize computations. As others have noted in specific instances [5], [8], the need to move data and synchronize is a manifestation of the *parallel data-flow limit* on a computation's performance. After aggressive precise optimization, value and convergence approximations, synchronization and data movement are the last impediments to improved parallel performance. Eliminating synchronization and data movement lead to inconsistency in both values and structures. We propose that SA is the key to overcoming the parallel data-flow limit. By selectively eliminating (i.e., by *approximating*) a program's synchronization and data movement and permitting inconsistencies, SA equips a system to overcome the final barrier to parallel performance.

II. THE PROMISE OF SA

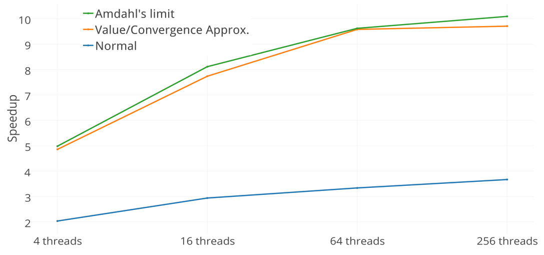
The purpose of this paper is to describe SA and to quantify its *potential* for improving parallel performance for a collection of data-flow-limited parallel applications. We conducted a limit study and our results are in Figure 1. In our study, we charted parallel speedup as core count increases, as predicted by Amdahl's Law. We compared that prediction to measured performance in a custom, Pin-based multi-processor simulator with several application configurations. The first configuration was a "normal", fully-precise execution, in which we execute all synchronization operations and all cache coherence operations, keeping all data precisely consistent.

We then applied a "limit case" of value approximation, whenever it was possible to do so without causing the application to crash. To implement value approximation, we removed *all* locks protecting data values whose imprecision did not cause a crash. We also eliminated the modeled cost of *all* cache coherence operations, simulating complete elision of coherence (which would leave values approximate, at least).

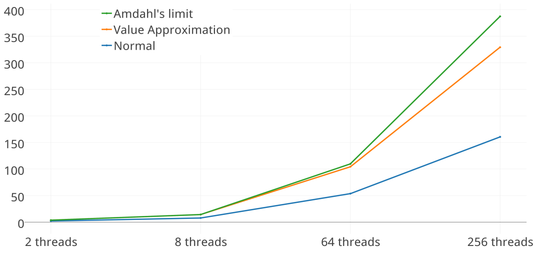
We applied convergence approximation to only one benchmark, K-Means [7]. To do so, we removed synchronization on data used to decide convergence, effectively eliminating the reduction operation that computes the application's convergence criterion. Eliminating the reduction is, in effect, a limit case of Paraprox's reduction approximation [9]. In addition to convergence approximation, we also applied value approximation, eliminating coherence and some value-only synchronization.

The results in Figure 1 illustrate several interesting connections between approximation and parallel performance. First, we note that in the Key-Value Store — and in the majority of numerical approximation targets in the literature [11], [6], [13], [3] — value approximation never causes a crash and, with no synchronization or data movement costs, the limit performance approaches the Amdahl limit. Second, we note that in K-Means, reducing results to determine convergence limits performance. Eliminating the expensive, synchronizing convergence computation eliminates serialization and waiting, pushing parallel performance scaling to the Amdahl limit.

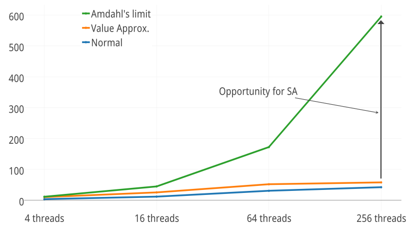
Third, and most importantly, we see that the performance of PageRank, SSSP, and BFS — even after value approximation



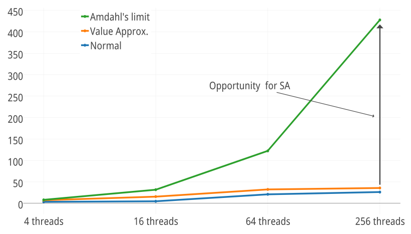
(a) K-Means



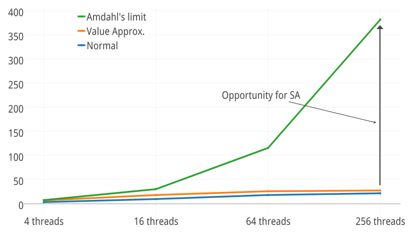
(b) Key-Value Store



(c) Page Rank



(d) SSSP



(e) BFS

Fig. 1: **Performance improvement in different applications due to approximations**

— is *far* from the theoretical limit. This performance gap is due to the synchronization left in these programs that we were unable to eliminate for this limit study. These synchronization operations are the ones that enforce *structural* correctness and they hamper parallel performance scaling, leaving it far below the Amdahl limit.

Unfortunately, without these structural synchronization operations, the applications crash because today's systems do not gracefully handle inconsistent data structures. To make these structural approximations viable, we need a new definition of data structure correctness that gracefully handles inconsistency without crashes.

III. The Need for a New Correctness Definition

Figure 1 shows the potential performance improvement of SA. A key facet to making SA a *useful* approximation mechanism is to develop a new definition of correctness that defines structural states permitted under SA.

Existing definitions of correctness under approximation are based on comparing the outputs of an approximate execution with one from a precise execution. We note that this correctness definition is useful only for value approximations.

We need to relax this definition of correctness to reason about SA executions that may have inconsistent data structures. The main reason we need such a relaxed definition is that existing definitions will typically *terminate* executions in many situations that stem from inconsistencies. For example, if an inconsistency leaves a link in a linked structure unassigned, an execution may terminate with a null pointer exception. A correctness definition for SA should permit inconsistencies by definition, even with, e.g., unassigned links.

With a relaxed correctness definition that permits inconsistencies, we envision a need to modify the execution model to accommodate structural inconsistency. One approach is to *do nothing*. This strategy is a gamble that, despite the inconsistency, executing code will make observations of and modifications to data structures that are reasonable. We expect that combining this approach with a quantification of error (e.g., number of unassigned links) may help determine when a structure is usable, or has become too inconsistent.

An alternative is to leverage data structure repair [2]. If a system using SA periodically, dynamically repairs broken data structures, it would reduce the possibility that structural inconsistencies are unrecoverable. One challenge to this approach is reducing the dynamic cost of repairing data structures and we expect that offloading repair computations to the cloud is a promising path.

Another possibility is to define data-structure-specific *resiliant operators* that do not cause a crash, even when working on a broken data structure. We expect resiliant operators to rely on tactics similar to manual defensive programming. Resiliant operators effectively broaden the set of allowed data structure states without failure, allowing aggressive SA. Combining resiliant operators with repair affords parallelizing repair work with the application's manipulations of the structure.

IV. Conclusion

In this paper, we presented a new form of approximation called Structural Approximation. We motivated the need for SA by quantitatively estimating the opportunity missed by existing approximation strategies. Our limit study results show that breaking the invariant that an application's data structure must be consistent at all times has potential for massive improvements. However, in order to ensure useful approximations, we identified the need to come up with new definitions of correctness, and proposed several directions for future work toward realizing SA.

References

[1] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam. Continuous real-world inputs can open up alternative accelerator designs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 1–12, New York, NY, USA, 2013. ACM.

[2] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 64–73, New York, NY, USA, 2007. ACM.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[4] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An online quality management system for approximate computing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 554–566, New York, NY, USA, 2015. ACM.

[5] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[6] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 309–328, New York, NY, USA, 2014. ACM.

[7] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188, Oct 2006.

[8] M. Rinard. Parallel synchronization-free approximate data structure construction. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[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] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 1, 2015.

[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] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. *ACM Trans. Comput. Syst.*, 32(3):9:1–9:23, Sept. 2014.

[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.