Compiler Support for Approximate Computing

Junhan Zhuo
Carnegie Mellon University
Email: junhanz@andrew.cmu.edu

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

I. Introduction

One of the major challenges in Approximate Computing is to target approximations which lie on the pareto-optimal boundary of performance and program quality. Compilers prove to be useful tools in finding *good* approximation targets that maximize performance while bounding the quality degradation of the program. In this project, we aim to develop a compiler analysis pass that finds approximation opportunities while providing some form of a guarantee on the error introduced into the program.

Research in the field of parallel computing has identified data movement as one of the major impediments to continued scaling of performance as the core count increases. Data movement is necessary for a program to execute *correctly*. However, a growing class of applications, like image processing, data mining and machine learning, are displaying a tolerance to errors in the program values. For these applications, we could limit the amount of data sharing to improve performance at the cost of small errors in the overall output of the program.

II. Idea

Most modern multi-core processors implement data movement among cores using cache coherence. In this project, we wish to explore the impact of reducing coherence operations on the performance of applications and the resulting quality degradation.

Our mechanism of cutting cache coherence is to partition the existing L1 cache in different cores into coherent and incoherent parts. The incoherent part of the cache will not enforce any form of coherence among multiple cores and, thus, houses data that is private to a core. The benefit of having private data is reduced communication among cores, which is essentially a serializing operation in a parallel execution.

While eliminating inter-core communication via data privitization is bound to improve the performance of parallel programs, we must be strategic about our decision as to when to privatize data since this decision controls the quality of the program. This is where we plan to utilize a compiler to identify approximation opportunities that will yeild performance improvement at the cost of insignificant error in the program output.

In order to estimate the potential damage due to approximation, we aim to develop a compiler pass that will analyse the impact of different variables used in a program. We will then mark data as approximable based on the frequency of the data reuse in the program. This pass will be described in greater detail in the next section.

III. Mechanism

As mentioned before, we need to strategically place data in the incoherent cache in order to bound the error in the program output to tolerable values. Our compiler analysis will search for shared data in programs whose correctness is not critical to program quality. In order to do this, we first need to identify shared data that can be approximated.

As an initial version of the project, we consider data that is guarded by synchronization operations. Prior research in approximate computing has shown that removing synchronization can be an effective way to explore the performance-correctness trade-off. We extend this concept by stating that if a synchronization operation is deemed unnecessary then providing coherence for the data value being guarded is also not important. In this manner, our compiler pass builds a set of candidates that can be approximated.

The next task is to select the data values from the above list of possible candidates that will not cause

significant quality degradation. In order to find such variables, we perform a liveness analysis of the candidates and determine the program lifetime of each variable. We use lifetime as a proxy for the importance of a variable on program quality. Variables with shorter lifetimes might not effect program quality much and vice versa. We aim to explore other dataflow analyses and metrics to find relative importance in subsequent versions of our pass. We are also considering passing a *quality threshold* to our pass which can then vary the amount of variables being approximated.

Since our proposal relies on hardware not found in existing architectures, we introduce ISA extensions that will place data in the incoherent cache. Our compiler pass will annotate the binary with these special instructions to convey to the hardware (simulated) that the data value must be placed in an incoherent cache. Subsequent accesses to the data value by a core will modify the data stored in its incoherent, private cache.

While the previous paragraph moving data from the coherent cache to the incoherent cache, we might also have to do the opposite at times. This could be a way to limit the accumulation of error value in the program. In order to get a coherent copy from at incoherent copy we apply an approximation based on the computation being performed in the original shared data. For example, if operation being performed on the original shared data was addition then we apply the updates of the addition to the local value of the core. When it is time to convert the local value to a global value, we randomly select the value from the incoherent cache of one of the cores and assume that the other cores also computed the same value. After performing this *approximate merge* we place the data in the coherent cache and from this point on all the cores see the data value.

In the initial version of our compiler pass, we will support only limited computations on shared data and their corresponding approximations (which will be similar in the spirit of [**?**]). We aim to implement more general operations in subsequent versions.

IV. Infrastructure

In order to compute the performance benefit of our approximation we will be using a PIN based cache simulator. The simulator adds fixed cycle costs based on the level of cache from which the data is accessed (three level cache modeled) and the coherence state of the cacheline (MESI protocol modeled). The simulation infrastructure is ready and has been in use for other research for quite some time.

As part of this project, we will model a part of each cores L1 cache as incoherent. Using PIN's routine instrumentation, we can identify the data that needs to be stored in the incoherent cache (stores only tag values). This way we can compare the execution time of the approximate version against the precise version to quantify the performance improvement.

In order to evaluate the error in a program due to incoherent manipulation of data, we will make the compiler convert shared data into thread private data. Based on when the programmer wishes to convert an incoherent data into a coherent data (through special functions/ISA extensions), we can randomly select one of the thread private data, apply the approximate merge and convert back to shared data. While we are actually duplicating data in the program, the logical view of the duplication is basically storing the data in an incoherent cache. The duplication of data allows us to run the effect of no coherence on a native machine that has coherence.

V. Project Plan

We have broken down the project plan into the following steps:

- Write a compiler pass to identify sychronization operations
- Design logic to identify the shared data that is guarded inside these synchronization operations and build a list of potential approximation targets
- Perform liveness analysis on these candidates and characterize variables on the basis of their lifetimes from the start of the approximation
- Apply the proper annotations (special functions identified by the simulator) for the data values that should be approximated. Also, convert the shared data into thread local data.
- Identify the programmer routine that indicates the application of approximate merge and add the code to perform the merge into the program's binary

- Run a generated binary on the simulator and native machine to get an idea of the performance vs error tradeoff
- Add support for more complex dataflow analysis for detection of approximation targets
- Add support for more computations that can be approximated

VI. Work Distribution

For the initial version of the compiler pass we will be working in tandem to set up the infrastructure. After the initial pass is complete, we will concurrently try to add separate features to the compiler pass.

VII. Project goals

We have set the following goals for ourselves:

- **75% goal:** Complete code annotation so that if we have any other heurestic for approximation then atleast the performance benefit of the approximation can be tested on the simulation infrastructure
- **100% goal:** Implement a liveness analysis based pass that detects approximation opportunity for a set of computations on shared data
- **125% goal:** Implement other dataflow analyses and test other quality metrics to detect approximation. Perform experiments to test the performance-accuracy tradeoff of different policies. Also, add support for more kinds of computations on shared data

VIII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.