Compiler Support for Restricting Data Movement

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

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

I. Project Motivation

The aim of this project is to improve the performance of parallel applications by trading off precision of the computations.

The scaling of parallel programs with increasing number of cores is limited by data movement within cores. However, this data movement - in the form of synchronization and cache coherence - is essential for consistent manipulation of shared program values. In this problem setting, approximate computing can help speedup the execution of parallel programs by reducing inter-core communication

Research in approximate computing is based on the observation that a growing class of important applications can tolerate imprecision in program values. The hypothesis here is that not all program values are equally important for a *good* program output quality. Using this hypothesis, we can reduce communication for unimportant values in parallel programs to speedup execution at the cost of negligible output quality degradation.

The task of finding the above mentioned *unimportant* values can be performed by a compiler. With the help of different data flow analysis, we can find locate the shared data of a program whose imprecision does not affect the program quality by much. In this project, we will be focusing our attention on reducing one aspect of inter-core communication - cache coherence - to improve performance of parallel programs while containing the output degradation of an application to a user specified threshold.

II. High level System Description

Most modern multi-core processors implement data movement among cores using cache coherence. In this project, we wish to explore the impact of cutting 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, private to each core, into coherent and incoherent parts. The incoherent cache will not enforce any form of coherence among multiple cores and, thus, houses data that is strictly 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 privatization will surely improve the performance of parallel programs, we must at the same time be strategic about our decision as to when to privatize data and when to merge private copies back. This decision controls the quality of a program. This is where we plan to utilize a compiler to identify approximation opportunities that will yield performance improvement at the cost of insignificant error in the program output.

At a high level, our optimization pass needs to do the following things:

- Identify a list of shared variables which, on approximation, will not cause the application to crash
- Prune the above list of candidates on the basis of dataflow analyses to meet user specified quality constraints
- Modify the application code to place data in an incoherent cache (In our case add special functions that will be understood by our custom PIN based simulator)

III. Implementation

In this section we present our plan for implementing the above goals and the progress we have made to that end.

We decided to break our optimization pass into three individual pass each handling a part of the above high level goals. The objectives of the three passes are listed below:

•

IV. Mechanism

As mentioned before, we need to strategically place data in the incoherent cache in order to contain 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 essential. 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* as an argument to our pass which can then statically 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 into 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 discussed 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 a program. In order to get a coherent copy from at incoherent copy we apply an approximation based on the computation being performed on the original shared data. For example, if the operation being performed on the original shared data was addition then we apply the updates of the addition to the local value in each 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 and update the local copy with a multiple of the total number of threads. After performing this *approximate merge* we place the data in the coherent cache and from this point on all, the cores see a coherent 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 [1]). We aim to implement more general operations in subsequent versions.

V. 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 shared 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.

VI. 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 the generated *approximate* binary on the simulator and a 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

VII. 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.

VIII. 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. Explore the sensitivity of the quality threshold on the resultant performance and error values.
- **125% goal:** Implement other dataflow analyses and test other quality metrics to detect approximationr. Perform experiments to test the performance-accuracy tradeoff of different detection policies. Also, add support for more kinds of computations on shared data

References

[1] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. 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). ACM, New York, NY, USA, 35-50. DOI=http://dx.doi.org/10.1145/2541940.2541948