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

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

The scaling of parallel programs with increasing number of cores is limited by data movement between 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. Structure of the Optimization pass

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:

- **Pass 1**: In this pass we identify shared data that can be approximated without causing program crashes
- **Pass 2**: For the variables identified in the previous pass, duplicate the shared data among different threads and change references to the shared variable by the thread private duplicate
- **Pass 3**: Modify the source code to indicate information that can be identified by the performance simulator

We shall now explain each pass in greater detail in the following subsections

A. Pass 1

The aim of this pass is to identify safe-to-approximate shared variables that are guaranteed to not cause a program crash.

As we are trying to find shared variables in parallel loops, the most straightforward approach is to find load-modify-store patterns in a loop, so we first located the place where there is a parallel loop, and within execution order find variables which have the load-modify-store pattern in the body of the parallel loop. As these variables are only used this way, they are some kind of accumulative variable which is related to the computation of the program rather then the control flow of the program and thus are generally safe-to-approximate variables. Later tests on these variables can be made to ensure that these are safe-to-approximate shared variables.

B. Pass 2

The aim of this pass is to model the error caused in the program output due to not providing cache coherence all the time.

We use the information of safe-to-approximate shared variables from the previous pass to duplicate the shared variables across different threads. The duplication of data allows us to understand the effect of not having cache coherence on a system that provides coherence at all times. In an original system, we would have had to perform a *merge* of the incoherent values to produce a single, globally visible value. We intend to perform a similar merge by making the pass place a merge function that applies a specific logic to produce a single value from multiple incoherent values.

C. Pass 3

The aim of this pass is to get an estimate of the performance benefit of not providing cache coherence at all times

Our simulator can be configured to identify certain function calls and acts accordingly. In this case at the place where we modified the code to not provide coherence, we added certain blank function calls that are useless to the program but helps the simulator indicate the part that we want to run without coherency.

IV. Progress

In this section, we present the progress we have made on our optimization pass

A. Pass 1

This pass first identified the body of parallel loops and then did a pattern matching for variables in the loop body with the load-modify-store pattern.

Identifying the parallel loop is not as simple as using LLVM's loop analysis pass, as not all loops are parallel loops such as initialization loops, what we did instead is used special indicators in the program to mark it as a parallel loop and when doing the pass to recognize it and then marking all the basic blocks in the parallel loop body.

As for the pattern matching, for each variable starting with a load we added it to the suspect list, and followed along the basic block, if later it inevitability and just have the modify and store operations without reaching any basic blocks not marked, then we added it to the recognized variables list.

B. Pass 2

This pass required using the shared variables' information from the previous pass to create thread private copies of the variables.

Functionalities implemented:

- Identification of the initialization point of shared variables in program bytecode
- Creation of duplicate copies for each shared variable

Functionalities to be implemented:

- Insert arbitrary merge function in the byte code that consolidates the incoherent copy to form a single value.

C. Pass 3

This pass takes the information of the previous pass indicating where it have changed and inserted blank function calls right before the modified code.

To make the compiler to not just optimize the function away, we did put irrelevant code in the function which access global data so that the optimizer won't optimize it away, but as this is irrelevant code in the simulator we can just configure it to ignore the function body when it encounters the function to not introduce another point of uncertainty.

V. Execution Plan

We have set up most of the initial infrastructure to test the performance and accuracy implications of providing coherence sporadically. The major roadblock for us till now has been to insert arbitrary function calls into the bytecode. We expect to resolve this issue soon and move on to implementing more functionalities to our optimization pass as mentioned in the proposal.

To reiterate our goals, we wish to have the following functionalities in our optimization pass by the final report:

- Incorporate support for identifying more synchronization primitives which, consequently, will allow identification of more approximation targets
- Apply a set of data flow analyses (primarily liveness analysis and reaching definitions) to prune the set of approximation in order to control accuracy loss
- Collect results for a range of application with different amounts of opportunities to apply approximations

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