Approximating Data Movement via Compiler Support

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

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

I. Introduction

Since the availability of multicore CPUs is widely introduced in the early 2000s, the concept of parallel computing have been widely accepted and extensively implemented. Developers developed parallel versions of algorithms, making uses of all the cores provided. But more often then not there are some shared data accessed and modified by more then one thread during the computation. This introduces data movement between separate core's cache, which recent research have identified as one of the major impediments to continued performance scaling with increasing core counts, which is a big issue in systems with big core counts like Tilera's TILE-mx100 SoC which delivers 100 ARM cores on a single chip.

On the other hand, approximating computing have became an emerging topic as a growing class of applications, like image processing, data mining and machine learning, are displaying a tolerance to errors in program values. By lifting the constraint of a fully precise result, approximate computing can offer substantial improved energy efficiency and deliver the resulted value much faster. This is ideal in circumstances like embedded systems where the provided computing power is quite limited but a result is needed on a real time basis like robotic motion scheduling, it is better to yield a sub optimal result as long as a result is given at every time frame then a optimal result which is delivered too late to be optimal anyways.

Recent research have been done in approximate computing in the fields of parallel computing, [1] tried to reduce synchronization between cores by removing locks, [2] tried relaxing the parallel program semantics like breaking barriers and relaxing sequentiality, [4] took another approach by reducing data accesses for some common patterns in parallel programs. But no matter if reducing synchronization or data accesses, they still have the issue of accessing the same shared data, which still have scalability issues. Thus in this paper we propose an architecture which have incoherent cache along with the regular coherent cache in the L1 cache private to each core. We then introduce special ISA extensions in dealing with our proposed incoherent L1 cache.

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 for finding *good* approximation targets that maximize performance while bounding the quality degradation of the program. In this paper, we implemented several heuristics as compiler analysis pass that suggests the approximation opportunities while simultaneously providing some form of a guarantee on the degree of error introduced to the program.

The rest of the paper is organized as follows: In Section II we explain our modeling of the architecture for supporting incoherent cache as well as the extended ISA instructions that we proposed. Section III explains how we implemented our compiler passes to identify the potential shared variables that can be approximated and heuristics to suggest how to approximate. Section IV shows our results for testing our heuristics against our toy application. And finally Section V presents our conclusion for this work and discussing limitations and potential future work.

II. Infrastructure

We propose an architecture where each core's L1 cache have an incoherent portion as well as a coherent part. We didn't limit the size of the incoherent part as well as the eviction and write back to the L2 cache as we are just doing a proof of concept here. We also propose two ISA extension instruction that are APPROX and MERGE. When the architecture hits the APPROX instruction it marks and put the provided argument into the incoherent cache, and each core's access and modify to the data only affects the local copy inside it's incoherent cache until it hits the MERGE instruction. The MERGE instruction acts as an implicit barrier where only when all cores who have the shared data reaches will the architecture does a merge of the incoherent copies of the data similar to the approximate version of reduction in [4]. After this point the data is in the coherent part of the cache and all subsequent uses of this shared data follows the MESI protocol.

We wanted to measure both the performance improvement and the potential error of our incoherent cache approximation, unfortunately there isn't an existing architecture or simulator which can evaluate such two fields at once. Though we could model an existing simulator to do this, the work would be too tedious as we are just doing a proof of concept, instead what we did was modeled and yielded two separate versions of the input program, one would be annotated with special functions calls that a simulator can recognize and account for the corresponding cycle count, and the other would be modified to take account for the effects of actually using incoherent caches. Even though they are two separate programs, they share the same idea of doing the same approximation of incoherent caching, just one is used to measure performance while the other accounts for the potential output error.

In order to compute the performance benefit of our approximation we used a PIN[3] 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.

For our purpose we modeled a part of each core's L1 cache as incoherent in the simulator, using PIN's routine instrumentation and identifying special instructions that we marked in the code (particular blank function calls which takes an argument that the simulator can then see marked as being stored in the incoherent cache), it can identify the data that needs to be stored in the incoherent cache and stores the tag values in the cache. We also inserted blank *do_ merge()* functions which tells the simulator the place to merge the incoherent data in the cache and adds the corresponding cycle counts. In the coherent part of the program the coherent value is used while in the incoherent part a separate incoherent value marked as mentioned above is substituted and used, as we are not worrying about the error in this program, we are only interested of the total cycle count after it's execution this approach is simple yet sufficient for us to use. To be noted as we are using incoherence caches, it makes no sense to keeping the synchronization guards aka locks guarding the shared data, so we didn't account for any cycle counts for the locking and unlocking of the critical sections guarding the shared data that we put into the incoherent cache.

In order to evaluate the error in a program due to incoherent manipulation of data, for each shared data to be made incoherent, we declared a separate array the size of the shared data multiplied by the number of threads used in the program (which is either given as a define or passed to the program as the first argument). Each thread spawned is given a unique tid going from 0,1... that is used to access it's local private copy of the shared data. In place where the APPROX instruction should be inserted the value of the data is copied into the separate declared array and in place where the MERGE instruction should be inserted an explicit merge is inserted to the code which puts the merged data back to the original data. The incoherence part of the data is used with each thread's own local copy of the data, while the coherent part remains the same using the original value. 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.

III. Compiler Analysis

In this project, we require compiler support to analysis the potential shared variables that could be made incoherent. We used version 3.5.0 of the LLVM compiler framework[5], and restricted our analysis to programs using the pthread library for parallel threading. We implemented our passes as middle end passes for the LLVM IR language.

A. Finding Potential Shared Variables

The first thing to be done is to find potential shared variables that can be approximated using incoherent cache. Shared variables should be stored in the memory and not allocated as just a core's register, so the first approach is to find variables with the load-modify-store pattern. We restricted our modify operations to be commutative so that the merge later would be straightforward. Second, the shared variable should be guarded within locks, and as we are restricting ourselves to pthread programs, we only have to seek out the *phtread_ mutex_ lock()* and *pthread_ mutex_ unlock()* function calls and assume that the control flow guarded by pairs of these two functions are in the critical section. So only access patterns of load-modify-store inside the critical section are considered candidates for approximation.

After finding the candidates we need to figure out the initial address of the candidates, the value given to the load instruction might be generated by a *getelementptr* instruction which calculate the offset address in an array, by another *load* instruction which loads the address in the case of a pointer pointer, or a mixture of both, so in our pass we have to find the initial value used to calculate the addresses, so if the address is generated by another instruction we keep reverse tracking until we hit a globally declared variable which we recognizes as the origin of the shared variable which we pass on to our further pass.

But before ending this pass we still have to do one more check, that is checking is the variable is used to determine the control flow of the program, as if approximating a shared variable into a value that should never be considered correct could result in non-deterministic control flow in the program and might result in a non-terminated program. Also in our simulator the approximated data is never calculated so if approximating the variable could lead to serious problems in our simulator. We implemented this by finding all use cases of the candidate and see if the value calculated using the candidate value is used in a branching statement or not, we can do a thorough analysis as to building a tree of all values that are affected by this value and see if any of them are used in a branching instruction, but this would be unreasonably complicated because normally programs aren't this complicated as going down just one or two levels down the tree is sufficient enough for most programs.

B. Heuristics

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. Thus we need compiler to do analysis to search for shared data in programs whose correctness is not critical to program quality. Given the list of potential candidates from the last pass, we need to provide heuristics on identifying data that when approximated estimated to not cause significant quality degradation.

1) Estimating Importance of Program Variables: Our first heuristics is to estimate the *importance* of the shared variables. We say the importance are correlated to the number of uses of the shared variable in the program. This can also build a tree of it's own including the uses of the values calculated using the shared variable, but for the sake of simplicity we only used the first level of uses of the shared data as a proxy to the importance of the shared variable.

We think variables that are used less might be better approximation targets (lead to lesser quality degradation). For example the scan operation which operates on an array will use the front data many times then the data at the end, and also the data in the front have a significant more impact on the overall quality of the scan operation while the last few data will only have influence on the last few elements after the scan operation.

For each data passed in from the previous pass this pass counts all the uses of each respective data, given the aggressiveness of the approximation in this case the threshold of the number of uses for a variable, it determines whether the variable should be approximated or not.

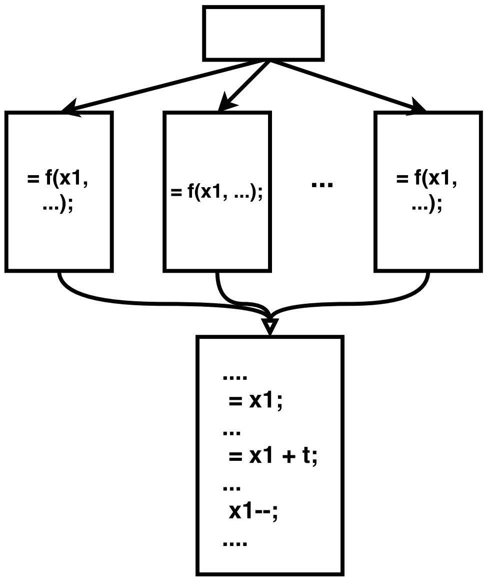

Fig. 1. Heuristic1: Variables that are used less might be better approximation targets

2) Quality impact of approximating variables: We think that the quality impact of a certain shared variable can be estimated by analyzing the place of use of the variable down the control flow. If the variable only appear on few paths, like only in one part of an unlikely taken branch path, then it is not likely to cause significant quality degradation on the program. More often then not the variable might not even be used by the control flow of many instances of the program execution. Else wise if a variable is used on all paths down the control flow then it certainly would have a quality impact on the given program.

We implemented this pass by not only counting the uses of the potential shared variables given, but also locate the use sites, that is in which basic block each use of the value is in. We constructed a separated set data structure for each basic block and put all the used variable in the corresponding basic block's set. We only counted the use of the load instruction and not the store instruction as sometimes the value itself is updated and sometimes it's used to figure out other values, and each instance should be counted as one, so by ignoring store instruction uses every instance of the use is only accounted for one time. After all the sets are constructed, we traverse through the control flow graph and see which variable is used only in certain path and which variable is used in all path. Given the aggressiveness of approximation, in this case the path frequency where the variable would be used, it determines which variables should be approximated or not.

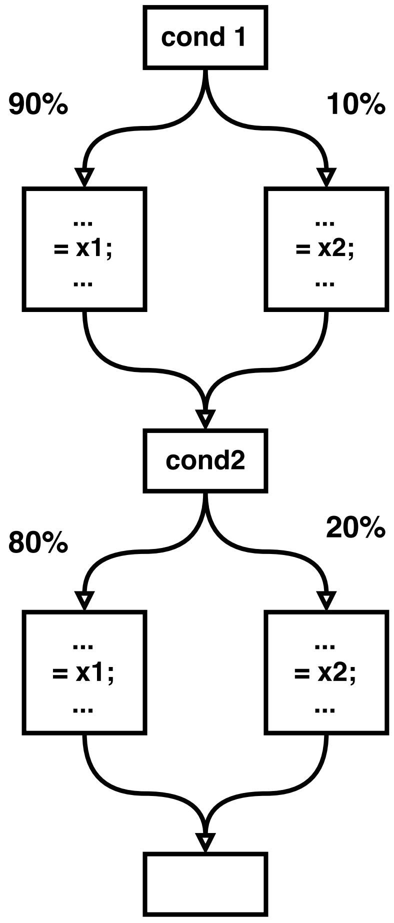

Fig. 2. Heuristic2: Variables that appear only on few paths are not likely to cause significant quality degradation

3) Detecting the Imbalance in Computation: @TODO Vignesh can you put something more here? Detect the amount of computation on variables in different threads.

This pass is implemented by first identifying all the calls to the *pthread_ create()* function and getting its third argument, which should be the function that the pthread created the thread to launch on. Then for every operations in the function counted how many uses of each of the shared variables, this serves as the indicator of the load of the current thread.

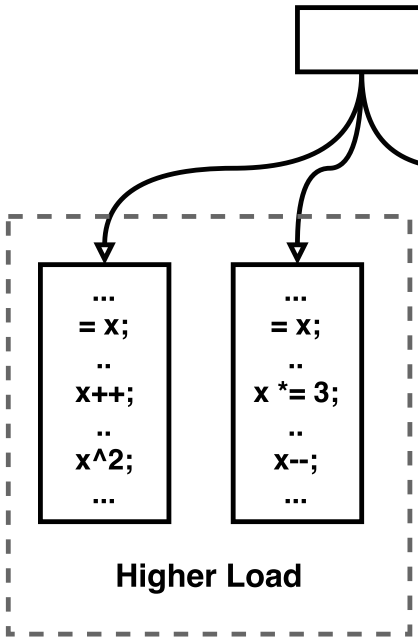

Fig. 3. Heuristic3: Variables that are heavily operated upon only in a few threads do not need coherence for all threads

IV. Idea

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 surelyo improve the performance of parallel programs, we must at the same time be strategic about our decision as to when to privatize data. 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.

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 on program quality. We will then mark data as approximable based on the frequency of data reuse in the program. This pass will be described in greater detail in the next section.

V. 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 [4]). We aim to implement more general operations in subsequent versions.

VI. 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.

VII. 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

VIII. 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.

IX. 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] L Renganarayana, V Srinivasan, R Nair, D Prener, C Blundell,Relaxing synchronization for performance and insight,Technical Report RC25256, IBM

[2] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization, in Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO 15), February 2015.

[3] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. 40, 6 (June 2005), 190-200. DOI=http://dx.doi.org/10.1145/1064978.1065034

[4] 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

[5] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04). IEEE Computer Society, Washington, DC, USA, 75-.