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

***Abstract*—Approximating computing is a growing domain as a growing field of applications tolerate degradation in the quality of the result in exchange for performance. And parallel programming is one of the many fields in approximating computing which is gaining attention. Parallel programs tends to have scalability issues due to high synchronization and data movement costs, while there are previous research to deal with approximating synchronization and restricting data accesses, none have done with approximate the cache coherence to limit data movement between cores. In this paper we are going to propose an architecture which have incoherent cache support, then we uses the compiler to statically analysis the program to determine the potential places the incoherent cache is put into use. Our evaluation shows that by using our heuristics we can achieve a decent speedup in performance (1.4x) whill maintaining a low introduced error ratio (<10%), and in some cases can even achieve a performance gain of nearly 2x without much quality degradation.**

I. Introduction

Since the availability of multicore CPUs is widely made 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. This is a big issue in systems with big core counts like the Tilera's TILE-mx100 SoC which delivers 100 ARM cores on a single chip[1].

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, [2] tried to reduce synchronization between cores by removing locks, [3] tried relaxing the parallel program semantics like breaking barriers and relaxing sequentiality, [5] 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 do a merge of the incoherent copies of the data similar to the approximate version of reduction in [5]. 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.

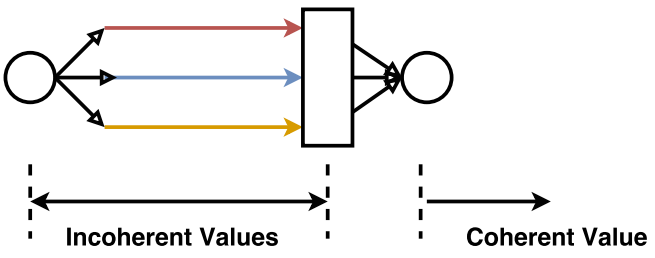

Fig. 1. Values are split into each core's incoherent cache after APPROX and merged back to the coherent cache at MERGE

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[4] 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 or wrapper 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 in 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[6], 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 address, 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 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.

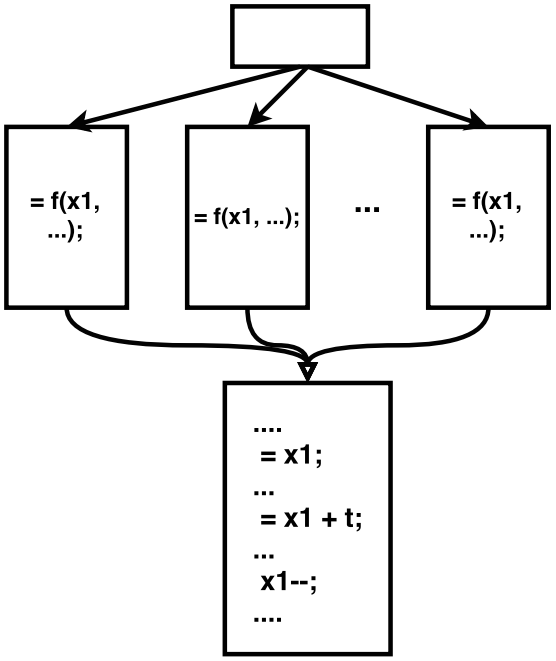

Fig. 2. Heuristic1: Variables that are used less might be better approximation targets

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, we can determine whether a variable should be approximated or not.

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. Elsewise if a variable is used on all paths down the control flow then it certainly would have a quality impact on the given program.

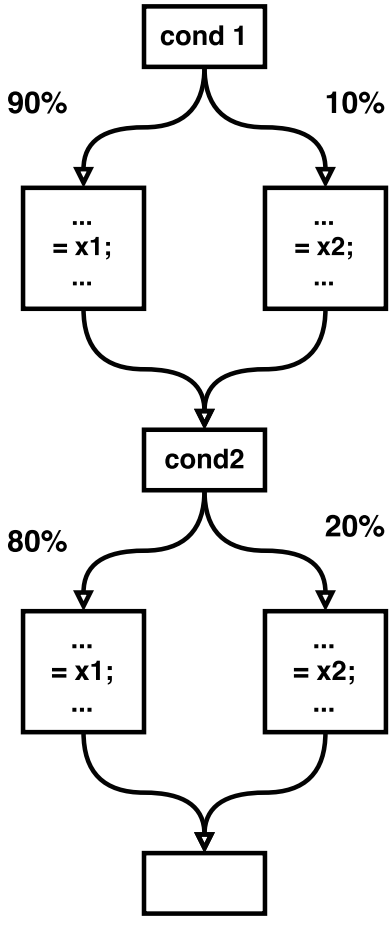

Fig. 3. Heuristic2: Variables that appear only on few paths are not likely to cause significant quality degradation

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, we can then determine which variable should be approximated or not.

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

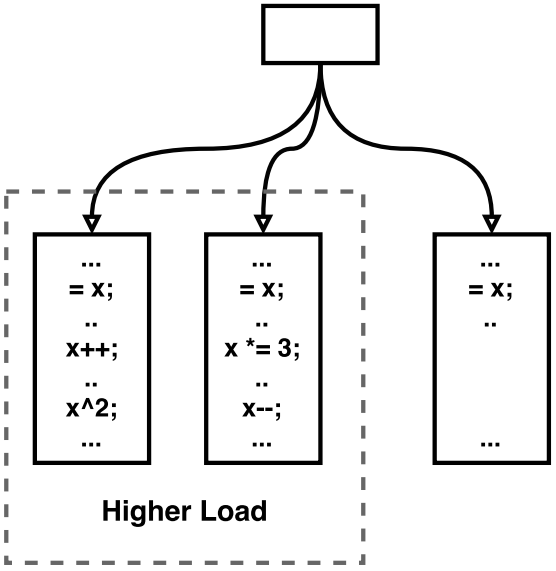

Fig. 4. Heuristic3: Variables that are heavily operated upon only in a few threads do not need coherence for all threads

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 work amount of the current thread.

IV. Evaluation

To evaluate the heuristics, we constructed several toy applications for our compiler passes to test out on. We modeled each of our toy application to correspond to a particular heuristic that we proposed. That is for heuristic 1 we generated a toy application where there are various random uses of the shared variables in the sequential portion of the program following the parallel portion (which is a common case for parallel programs, like in the map-reduce structure there is a massive parallel *map* stage followed by a sequential *reduce* stage), and for heuristic 2 we generated the sequential portion to include lots of different control flows with a lot of if-else structures. For heuristic 3, as parallel computations is not just single instruction multiple data (SIMD) type, we gave the *pthread_ create()* for each thread a different work function with different work loads inside each of them.

We evaluated our results by first running the original application directly for its intended output, then gave it to our PIN simulator to see the cycle counts for running this application. After having the results for the precise version of the program, we gave it to our shared variables finding pass and passed it along to its corresponding heuristic pass. Upon using the information from the heuristic pass we tested various levels of approximation aggressiveness generating the various approximated version of the program into the performance and error evaluation version. We then gave the performance version to our simulator to get the cycle count of the approximated version and run the error version to see the approximated result and finally compared it with the results of the original version.

The results of our evaluation can be seen in figure 5. As we can see, the performance ...@TODO Vignesh can you put in the results for the performance and error here?

@TODO Vignesh, can you also discuss a bit about the results here? Like the reasons for the performance and error gains? What is considered acceptable etc?

V. Conclusion

As synchronization approximation have been tried before, we took one step further and also

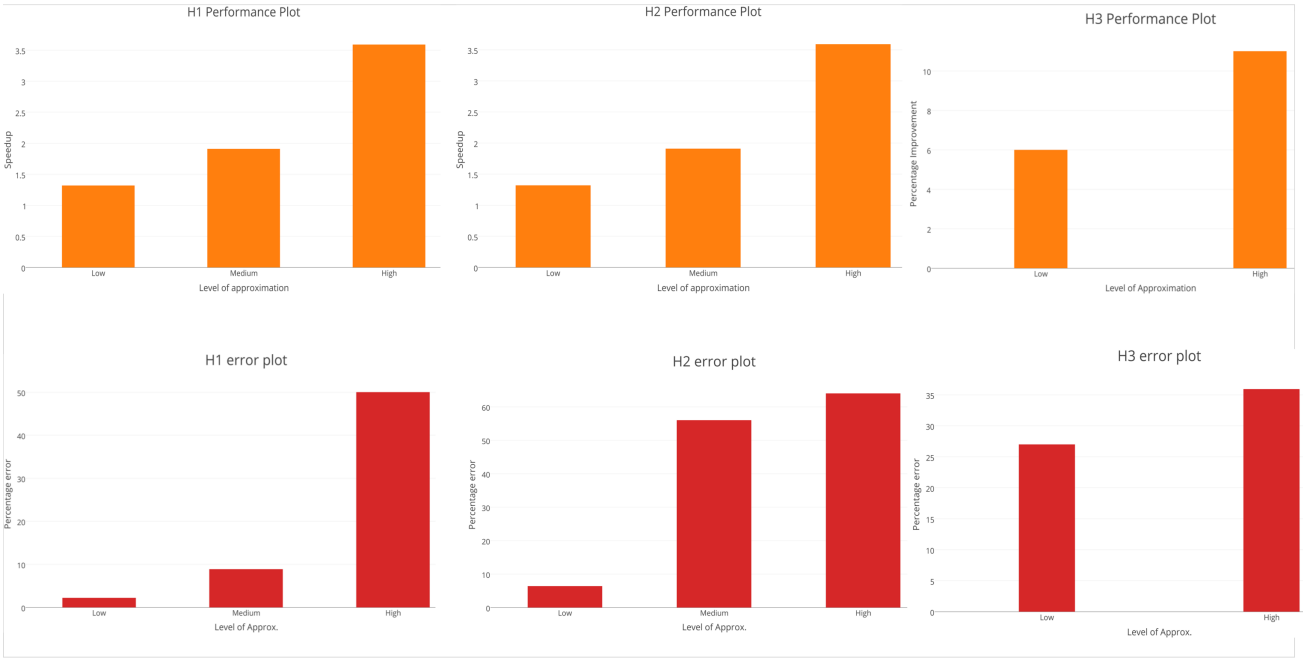

Fig. 5. Evaluation results on performance and error for the three proposed heuristics

approximated the shared variable guarded by synchronizations by proposing our own cache incoherent model of the architecture. And in order to find potential good values to approximate, we built compiler passes that first identified the shared variables and then applied heuristics to estimate the potential impact of the variable with uses count, path appearance estimate and imbalance between threads. Our results showed a decent improvement for our toy applications while sustaining minimal error rate if not approximated too aggressively.

Limitations on our approach is as we are only doing a static analysis of the program, we have no way of knowing any run-time dynamic information. Other systems such as [7][10] have done code profiling to get an idea on the dynamic aspects of the code, [2][3][5][7] all used run time frameworks to dynamically adjust the approximating rate based on the resulting error rate, rewinding and redoing the precise version of the program if the approximate version's result is not ideal ensuring there is a upper bound on the quality degradation of the approximated program. Future work can be done here to also build our own run time framework or integrate with one of the available frameworks to provide the dynamic analysis capabilities and rewind on unacceptable error ratio options that we lack now.

Future work can also be done by letting the programmer write special code annotations to tell our compiler which variable is a good candidate for approximation as in [8][9][10]. The programmer doesn't need to indicate each and every variable to be approximated, by just telling a potential candidate future work can be done using inductions in the ways of [10][11] tracking the uses and users of the given variable to find a good approximate candidate.

VI. Contributions

Junhan Zhou: 50%
Vignesh Balaji: 50%

References

[1] Bob Doud, Accelerating the Data Plane With the TILE-Mx Manycore Processor, http://www.tilera.com/files/drim__EZchip_LinleyDataCenterConference_Feb2015_7671.pdf

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

[3] 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.

[4] 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.

[5] 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.

[6] 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-.

[7] Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. SIGPLAN Not. 45, 6 (June 2010), 198-209.

[8] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. 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)

[9] Sampson, Adrian, et al. "Accept: A programmer-guided compiler framework for practical approximate computing." University of Washington Technical Report UW-CSE-15-01 1 (2015)

[10] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. 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)

[11] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision, in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 14), June 2014.