Static Reuse Profile Estimation for Stencil Applications

Abdur Razzak[1], Atanu Barai[2], Nandakishore Santhi[2], Abdel-Hameed A. Badawy[1,2]
[1] Klipsch School of ECE, New Mexico State University, Las Cruces, NM 80003, USA,
[2] Los Alamos National Laboratory, Los Alamos, NM 87545, USA
{arazzak, badawy}@nmsu.edu, {abarai, nsanthi}@lanl.gov

ABSTRACT

Reuse distance analysis is a widely recognized method for application characterization that illustrates cache locality. Although there are various techniques to calculate the reuse distance profile from the dynamic memory trace of a program, it is both time and space-consuming due to the requirement of collecting dynamic memory traces from runtime. In contrast, static analysis-based reuse profile estimation is a promisingly faster approach since it is calculated in the compile time without running the program and collecting the traces. In this work, we present a static analysis technique to determine the reuse distance profile of primarily loop-based programs. For an input program, we generate a basic block-level control flow graph and the execution count by analyzing the LLVM IR file. We present the memory accesses of the application kernel in a compact bracketed format and use a recursive algorithm to predict the reuse distance histogram. Under the hood, we deploy a separate predictor that unrolls the loop for smaller bounds, generates a temporary reuse distance profile for those small cases, and using these profiles, predicts the actual loop bound. Results show that our tool can predict 95% on average of cache hit rates, performs quite similar to the dynamic predictor, and is certainly much faster than dynamic approaches.

1 INTRODUCTION

In this era of high-performance computing, where microchips consist billions of transistors, it is important for applications to utilize this immense computing power. Therefore, performance modeling and application profiling play a significant role in helping us to understand how better a computing system is being utilized. A reuse distance histogram is a popular metric to measure an application's efficient memory utilization performance. It represents an overall picture of how often the same memory is used. It will also help to estimate cache hit rates as a metric of program performance. A higher cache hit rate indicates lower execution time for the program because it significantly reduces the number of times a cache line needs to be fetched from the main memory.

Reuse distance (RD) represents the number of unique memory accesses between two consecutive accesses to the same memory address. The reuse distance profile is the histogram of the reuse distances for all the memory accesses of a program. It has been an interesting topic for researchers to calculate reuse distance histograms correctly and in a faster way. They proposed both dynamic and static approaches. In the dynamic technique, dynamic memory traces are collected by running the program and then the calculation process begins. PARDA [7] calculates reuse profile from dynamic memory trace using a multi-process architecture. ReuseTracker [8] is a recent investigation on the parallelization of the dynamic reuse profile calculation approach. Even though these tools are faster and accurate, these tools require program execution and trace collection which consumes a large amount of time and disc space. On the contrary, static approaches are faster compared to dynamic approaches as it does not require runtime trace collection. [1, 2, 4, 6] algorithms are significantly faster to calculate reuse distance histogram. However, they are limited in terms of supported programming language. On the other hand, our approach is LLVM compiler infrastructure based and thus platform independent and supports a wide array of programming languages supported by the LLVM project. A large portion of modern program kernels consist of stencil loops. All these factors motivated us to develop methodologies to predict reuse profiles for nested loops and arrays of program kernels statically. Figure 1 helps to understand the problem more.

```
1       int i, j, k, alpha = 99;
2       int tmp[50][50], A[50][50], B[50][50];
3
4       for (i = 0; i < 30; i++)
5         for (j = 0; j < 20; j++)
6         {
7           for (k = 0; k < 10; ++k)
8             tmp[i][j] += alpha * A[i][k] * B[j][k];
9         }
```

Figure 1: Example code of nested loop and arrays

In this work, we introduce a static approach to estimate reuse profiles using LLVM intermediate representation (IR) using our calculation method. LLVM [5] provides a large array of static code analysis tools to characterize programs such as basic blocks with variable references inside, each basic block execution time, operation sequences, and a lot more. With the help of this tool, we can have an intermediate representation of basic blocks and generate a sequence of them. From there, we have the information of variables when they might be executed. Using our static calculator, we can produce some smaller loop bound examples, and then, by analyzing them, we can predict the bigger bound. In this way, we can achieve independence in the loop bound.

In the beginning, we get the LLVM IR file of the program using the Clang compiler. To read the LLVM IR file, the correspondent reader was utilized. It generates the static program trace. Then, using a trace analyzer, the static trace is analyzed and produces the basic Block Control Flow Graph (CFG). Each basic block of the program is represented as a node in the graph. After that, a CFG analyzer takes the graph as input, using memory use reference, and branch probabilities, it represents the whole program as a loop annotated trace which is readable and shows the loop bounds and array positions. At this point, we can not calculate the array references with a loop because the variables change the array position. To get a full picture of array reference changing, we have flattened the traces for smaller loop bounds such as 2, 3, and 4. From that flattened trace, now we can calculate the reuse profile for the reuse profile for those smaller loop bounds. Then, we have to reach the given bound in the program, and therefore, we have developed

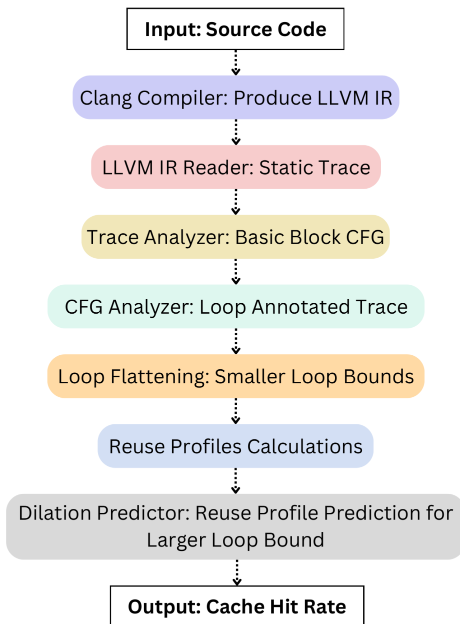

Figure 2: Steps of static analysis based reuse profile prediction

a static predictor that observes the changes in the reuse profile from 2 to 3 and 3 to 4. Based on the analysis, the predictor then generates some mathematical relationship between the number of those profiles and after having the equation, it predicts the n bound of loop.

2 RELATED WORK

Researchers are trying to find a more optimum solution for the reuse profile calculation.

Among the static techniques, Narayanan *et al.* [6] tried to calculate the reuse profile by using a syntax tree. However, the array scenario inside a loop is not supported by this technique. Meng-Ju Wu [9], investigated reuse distance on multi-core cache hierarchy scalability. They also showed that interference-based locality degradation is more significant than sharing-based locality degradation. However, their models have increased the amount of cache misses. Bush *et al.* [3] introduced some real-world programs that have in static approach, however, they are limited to only a few applications.

PARDA [7] proposed the first-ever effort to parallelize performance analysis with Reuse Distance. They divided their memory trace into multiple processors and applied a tree-based algorithm to calculate reuse distance. Due to the optimization, they achieved 13 to 50 times speedup and full accuracy. However, they have to wait until the runtime trace collection process is completed.

Our method takes a probabilistic approach, utilizing LLVM IR to calculate reuse profiles and other program metrics. This strategy allows us to accommodate multiple programming languages supported by the LLVM compiler infrastructure. Importantly, it generates precise profiles without requiring modifications to the source code. Furthermore, our outputs are rigorously tested against advanced dynamic models to ensure accuracy and reliability.

3 METHODOLOGY

This section elaborates on our sub-modules of static analyzers in detail. The info-graphic workflow is explained in the Figure 2. Our tool takes a source code program as an input and after doing all the processing, it generates a cache hit rate. In each step, we describe the context based on the example code given in Figure 1.

3.1 Produce LLVM IR using Clang Compiler

First, using the command **clang -g -c -emit-llvm 2mm.c**, we create LLVM Intermediate Representation of the source program from the Clang Compiler. As a compiler, clang executes pre-processing and compilation steps. Usually, it stores LLVM IR in a binary format, also known as bit-code, and generates the final executable file. However, since we have instructed to generate the middle step file by saying **-emit-llvm** for the 2mm.c file, it outputs the IR instead. It also includes the debug information.

3.2 Static Trace from LLVM IR Reader

In this step, the IR reader takes input from the LLVM IR file, reads all over the program, and produces all the basic blocks including the variable and instruction included inside, entry and exit of functions, all other labels and derails of load-store or branch instructions and a lot more. Arrays are identified as a list of GEP addresses and we can enlist how many arrays are there with their dimensions. We can also find the loop variables. If an array is inside the loop it shows memory references, however, at this point, we can not say which loop variable is accessing the array. It will be decided at a later stage.

3.3 Trace Analyzer

There are two sub-parts of the Trace analyzer.

A) Producing Control Flow Graph (CFG): The main goal of the task is to generate a graph with the specific information of parent and child nodes. Each node is an independent Basic Block. The correspondent memory accesses are kept on the same block one by one. Every successors and predecessors are marked. Therefore, each node knows what are the probable destination nodes. Each vertex of that graph is directional and represents source and destination nodes. The vertex are weighted and it represents the probability of going that path.

B) Basic Block Execution Counts :

Since we already have the Control Flow Graph completed at the previous stage, now we exactly know the execution count of a basic block. We can assume that all the memory references inside that basic block will be executed at the same number of times of the execution count.

3.4 Loop Annotated Trace from CFG Analyzer

At this moment, we have the CFG, branch probabilities, direction with Basic Blocks, and their execution counts. It looks like same as the memory reference, however, we can generate a virtual trace. We also have the edge probabilities in the CFG. Using all that information, we can calculate the path with maximum probability. By selecting that path, we can separate the loop or circular connection.

We can mark the loop starting, and ending boundary and also with a loop bound. After this processing, we have the below output.

However, we have to mark the arrays with their loop variable, or else we can not keep track of the memory reference to calculate the reuse. Manually we added a loop variable with the array. We search for the proper GEP name of the array, then we looked for the IR file again to identify the dimension of the array. Usually, the array controlling variable has a load call before the array, if it is not immediately accessed back to back. For example, in the trace, arrayidx8 is two dimensional in the IR, therefore, previously accessed 2 variables are this array's loop controlling variable. arrayidx8[i][k] in this example. After updating, the new trace looks like as following.

retval → alpha → i → [30 → i → j → [20 → j → k → [10 → k → alpha → i → k → A~i~k → j → k → B~j~k → i → j → tmp~i~j → tmp~i ~j → k → k →] → k → j → j →] → j → i → i →] → i

3.5 Loop Flattening: Smaller Loop Bounds

Since we have to calculate for big loop bounds and the array is involved inside the loop. we can not predict in a static way until we see some smaller complete scenarios. For example, in the above example, the 3 nested loop bounds are 30-20-10 and the loop variables are i-j-k. It is very time-consuming to produce all flattened scenarios for the full-length loop bound [figure 3 explains the process]. At first, we make 2-2-2 loop bounds as our baseline and flatten the whole loop. It looks as follows for the first i

['i', 'j',
'j', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-0', 'j', 'k', 'B~j~k-0-0', 'i', 'j', 'tmp~i~j-0-0', 'tmp~j-0-0', 'k', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-1', 'j', 'k', 'B~j~k-0-1', 'i', 'j', 'tmp~i~j-0-0', 'tmp~i~j-0-0', 'k', 'k',
'k', 'j', 'j',
'j', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-0', 'j', 'k', 'B~j~k-1-0', 'i', 'j', 'tmp~i~j-0-1', 'tmp~i~j-0-1', 'k', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-1', 'j', 'k', 'B~j~k-1-1', 'i', 'j', 'tmp~i~j-0-1', 'tmp~i~j-0-1', 'k', 'k',
'k', 'j', 'j',
'j', 'i', 'i', ...]

Now, we calculate flattened traces for 2-2-3 to observe how they change in the reuse profile and mark the impact of increasing 1 k. In the same way, from analyzing the 2-3-2 profile, we observe the increase of 1 j. After that, we are interested to see 2-3-3 when they both increase by 1. Here we will try to find a coefficient of j and k increase. After building a connection from these small bounds, we want to reach 2-20-10 bound. This process will be explained in the dilation section.

3.6 Reuse Profile Calculation

Then we calculate the reuse profile for those flattened stack traces individually. The reuse profile is the histogram of reuse distance whereas reuse distance is the unique reference between two same memory access. We have used the Least Recently Used technique to keep track of the most recent use of the same addresses. If the item is not found in the LRU, that means it is a cold miss or the first time accessing the address. Then we put that address into the LRU with the index. Otherwise, if it is found on LRU, then we take a hash set between the current index and the LRU stored index. That filters out the duplicate references in the middle. We stored the count of other addresses in a dictionary. That dictionary turns into a reuse profile. The algorithm 1 shows the process of calculation.

Algorithm 1 Calculating Reuse Profile

```
1:  procedure CalcReuseProfileFlattened(stack_trace)
2:      rf ← {}
3:      inf ← [ ]
4:      LRU_dict ← {}
5:      for index, item in enumerate(stack_trace) do
6:          if item not in LRU_dict then          ▷ If the item not found
7:              inf.append(item)
8:              LRU_dict[item] ← index
9:          else
10:             sub_list ← stack_trace[LRU_dict[item] + 1 : index]    ▷ Take all subset between two refs
11:             unique_items ← list(set(sub_list))    ▷ Remove duplicates
12:             reuse_distance ← len(unique_items)
13:             if reuse_distance in rf then
14:                 rf[reuse_distance] ← rf[reuse_distance] + 1
15:             else
16:                 rf[reuse_distance] ← 1
17:             end if
18:             LRU_dict[item] ← index    ▷ Updating with the least recently used index
19:         end if
20:     end for
21:     return rf, inf
22: end procedure
```

3.7 Dilation Prediction

Up until now, we have the reuse profile for loop bounds 2-2-2, 2-2-3. From these two reuse profiles, we determine the reuse profile for larger loop bounds. We have marked the increment in each individual reuse distance by increasing one k. Similarly, we have applied the same technique for 2-2-2 to 2-3-2 and observed likewise for increasing one j. Then, we compare the reuse profile of 2-2-2 and 2-3-3 and mark the change for the increasing of both j and k. After having the coefficient here, now we apply the following equation 1 to reach the J and K's target bounds such as 2-20-10.

$$RP = B_{22} + Inc_J \times U_J + Inc_K \times U_K + Coff_{JK} \times U_J \times U_K \quad (1)$$

In this equation, RP represents the frequency of the Reuse Profile; B_{22} is for the baseline reuse distance, mainly 2-2-2 is considered as the baseline reuse profile; Inc_J is the frequency increase for each increment of the loop bound J; U_J is the distance that J is going up;

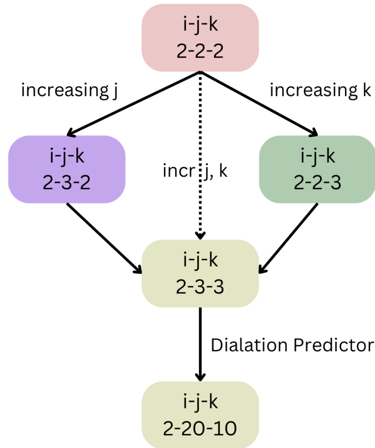

Figure 3: Loop flattening for smaller bound.

Table 1: List of applications used to validate our results.

App	Domain	Loops and Bounds (outer to inner)
2mm	Linear Algebra (LA)	3 nested; 300-200-100
3mm	LA	3 nested; 30-40-50
atax	LA	2 nested; 150-70
mvt	LA	2 nested; 100-200
gemver	Basic Linear Algebra Subprograms	2 nested; 400-500

It is also same for the K loop bound in the next term, and $Coff_{JK}$ is the impact factor of scaling both J and K bounds together.

By following this rule, we can calculate 2-J's bound-K's bound (such as 2-20-10), and then from there, using the same process, we can reach the I's bound (such as 30-20-10) or go more upper one by one loop.

4 RESULTS

Our investigation of finding a static way to calculate reuse profile and cache hit rates performs quite similarly compared to the well-established dynamic reuse profile calculators in terms of hit rates. We have validated our results using [7]. The same program is tested in both calculators and using the generated reuse profile, predicted cache hit rates as well using the same tool. The benchmark applications are analyzed in both frameworks. We have generated the cache hit rate from the reuse profile we have got and the results are shown in the diagrams. In the X axis, 5 different programs are mentioned and the Y axis represents the cache hit rates in the range of 0 to 100 scale. On average our tool can hit more than 95% which is significant in terms of static process.

Cache Reference: To predict the cache hit rate, we have used a cache where the associativity is 20, 20, and 8 for the first, second, and third level cache respectively. Also, Cache sizes are 16 MB, 25 MB, and 32 KB. However, the cache line is 64 for all 3.

5 LIMITATIONS

There are a few limitations to this approach. Calculating from bottom to top works only for a few numbers of nested loops. When the loop depth increases, the coefficient value does not work as expected. Therefore, discrepancies might show up in the reuse profile which could be a potential future work.

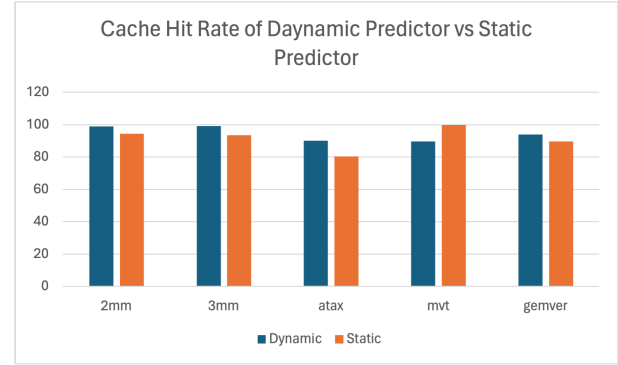

Figure 4: Comparison in Hit Rate between the dynamic and static predictor.

Another issue is this technique also does not have coverage for the if-else branch inside a loop. It only predicts a bigger loop-bound reuse profile from smaller cases, however, if any branch condition appears, predicting the reuse profile statically would be challenging, because, branches may not follow the same sequence of calculation.

6 CONCLUSION

Application profiling is significant for performance measurement. Reuse profile and cache hit rates are very important metrics. This paper has shown a unique technique to calculate reuse profiles by gathering knowledge from the smaller cases. It is independent in terms of loop bound and also a faster approach because the full process follows a static process. As this approach does not require the collection of dynamic traces, it saves time on the overall calculation process compared to the dynamic reuse profile predictors who collects the trace. The result shows that our tool almost hits the cash more than 95% on average, Sometimes it earns more hits than the dynamic predictor. This technique might be helpful when time is more important than compromising little accuracy.

REFERENCES

[1] Atanu Barai, Nandakishore Santhi, Abdur Razzak, Stephan Eidenbenz, and Abdel-Hameed A. Badawy. 2024. LLVM Static Analysis for Program Characterization and Memory Reuse Profile Estimation. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, VA, USA) *(MEMSYS '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/3631882.3631885

[2] Kristof Beyls and Erik H. D'Hollander. 2009. Refactoring for Data Locality. *Computer* 42, 2 (2009), 62–71. https://doi.org/10.1109/MC.2009.57

[3] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. [n. d.]. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience* 30, 7 ([n. d.]), 775–802.

[4] Arun Chauhan and Chun-Yu Shei. 2010. Static Reuse Distances for Locality-Based Optimizations in MATLAB. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. Association for Computing Machinery, New York, NY, USA, 295–304. https://doi.org/10.1145/1810085.1810125

[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* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–86.

[6] Sri Hari Krishna Narayanan and Paul Hovland. 2016. Calculating Reuse Distance from Source Code. (1 2016). https://www.osti.gov/biblio/1366296

[7] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. 2012. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 1284–1294. https://doi.org/10.1109/IPDPS.2012.117

[8] Muhammad Aditya Sasongko, Milind Chabbi, Mandana Bagheri Marzijarani, and Didem Unat. 2021. ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer. *ACM Trans. Archit. Code Optim.* 19, 1, Article 3 (dec 2021), 25 pages. https://doi.org/10.1145/3484199

[9] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 499–510. https://doi.org/10.1145/2485922.2485965