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 similarly 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

```
1       int i, j, k, alpha = 99;
2       int A[500][500], B[500][500];
3
4       for (i = 0; i < 300; i++)
5         for (j = 0; j < 200; j++)
6           for (k = 0; k < 102; k++){
7                 A[i][k] = alpha * A[i][k];
8                 B[j][k] = alpha;
9           }
10 }
```

Figure 1: Example code of nested loop and arrays

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 histograms. 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.

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 obtain the LLVM IR file from the program using the Clang compiler. To read the LLVM IR file, the corresponding reader is utilized, which generates the static program trace. Then, using a trace analyzer, the static trace is analyzed to produce 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 and, using memory references and branch probabilities represents the entire program as a loop-annotated trace, which is readable and shows the loop bounds and array positions. At this point, we cannot calculate the array references within a loop because the variables change the array position. To get a complete picture of array reference changes, we flattened the traces for smaller loop bounds, such as 2, 3, and 4. From that flattened trace, we can now calculate the reuse profile for those smaller loop bounds. Then, we must reach the given bound

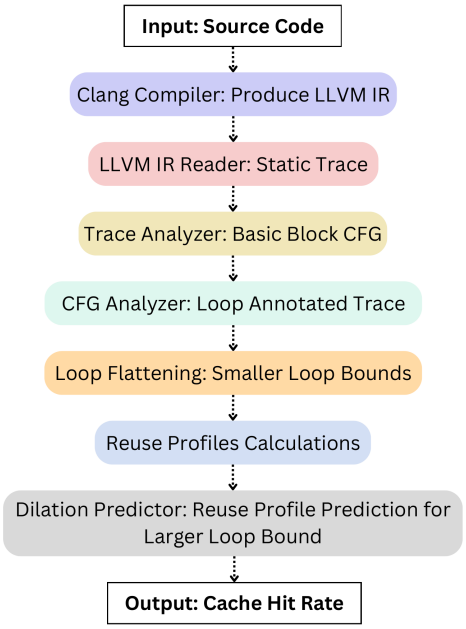

Figure 2: Steps of static analysis based reuse profile prediction

in the program, and therefore, we developed a static predictor that observes the changes in the reuse profile from 2 to 3 and from 3 to 4. Based on this analysis, the predictor generates a mathematical relationship between the number of profiles, and after deriving the equation, it predicts the n-th bound of the 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 a static approach, however, they are limited to only a few applications. Barai *et al.* [1] proposed a reuse distance calculation method utilizing LLVM-based techniques, but they cannot calculate reuse profiles for array-based programs.

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 the steps 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 that contain the variables and instructions 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 its probable destination nodes. Each vertex of that graph is directional and represents source and destination nodes. The vertex is weighted, and it represents the probability of going that path.

B) Basic Block Execution Counts :

Now that we have completed the Control Flow Graph in the previous stage, we know the exact number of times each basic block will be executed. This allows us to assume that all memory references within a basic block will be executed the same number of times as 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 the 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 boundary of the loop starting position, and the end to mark it as a loop bound area. 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 searched for the proper GEP name of the array, and 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 a two-dimensional array in the IR, therefore, previously accessed two variables are this array's loop-controlling variable. arrayidx8[i][k] in this example. After updating, the new trace looks like as following. Barai *et al.* [1] also followed similar steps until this, but we differ from their approach in the next steps in predicting reuse profiles for array-based programs.

retval → alpha → i → [300 → i → j → [200 → j → k → [102 → k → alpha → i → k → A~i~k → i → k → A~i~k → alpha → j → k → B~j ~k → 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 and complete scenarios. For example, in the above example, the three nested loop bounds are 300-200-102, 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 5 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', 'i', 'k', 'A~i~k-0-0', 'alpha', 'j', 'k', 'B~j~k-0-0', 'k', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-1', 'i', 'k', 'A~j~k-0-1', 'alpha', 'j', 'k', 'B~j~k-0-1', 'k', 'k',
'k', 'j', 'j',
'j', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-0', 'i', 'k', 'A~i~k-0-0', 'alpha', 'j', 'k', 'B~j~k-1-0', 'k', 'k',
'k', 'alpha', 'i', 'k', 'A~i~k-0-1', 'i', 'k', 'A~j~k-0-1', 'alpha', 'j', 'k', 'B~j~k-1-1', 'k', 'k',
'k', 'j', 'j',
'j', 'i', 'i', ...]

Now, we generate 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 try to find a coefficient of j and k increase. After building a connection from these small bounds, we want to reach 2-200-102 bound. This process will be explained in the dilation section.

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
```

3.6 Reuse Profile Calculation

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. 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.

3.7 Dilation Prediction

Following the previous steps, we can calculate reuse profiles for smaller basic loop bounds. However, from these smaller reuse profiles, we need to calculate the reuse profile for larger loop bounds as shown in the example program in figure 1. To achieve this target, even though our target, we step by step address from the inner loop bound to the upper loop and go further upper.

3.7.1 ***Prediction of Change in Reuse Profile due to Scaling Inner Loop Bounds:*** In this step, we only consider the most inner loops where the array's references are only changing with the one loop variable keeping the other positions constant. In the figure 3, an example code snippet is shown where both 2-dimensional arrays A and B are using a constant 0 for the first dimension and k is the

only loop variable that goes from 0 to 101 which is used as the second dimension.

```
1    for(k=0; k<102; k++) {
2        A[0][k] = alpha * A[0][k];
3        B[0][k] = alpha;
4    }
```

Figure 3: Array references are changing a single loop.

In most multi-level loop-based programs, the most inner loop will be repeated for each outer loop run. Therefore, if we can predict the scenario of the inner loop bound complete run, we can later anticipate the outer bounds by multiplication.

Now, if we produce a few loop annotated traces and reuse profiles for the smaller bounds starting from 2, we can create a relation between the numbers of each reuse distance. The loop annotated trace for the figure 3 looks like the following if $k = 2$.
[retval, k, [2, k, alpha, k, A~0~k, k, A~0~k, alpha, k, B~0~k, k, k,], k]]

In the same way, compute the reuse distance histogram for $k = 2, k = 3$, and $k = 4$ using the loop flattening technique that is explained in section 3.5, we may find the reuse profiles shown in table 1.

K (Inner Loop Bound)	Reuse Distance Histogram
2	{0: 5, 1: 9, 2: 5 , -1: 7}
3	{0: 7, 1: 13, 2: 8 , -1: 9}
4	{0: 9, 1: 17, 2: 11, -1: 11}

Table 1: Reuse profiles for smaller cases of one loop

From these smaller cases, we observe the number sequence and find a linear connection between the bound k and reuse distance frequency increment. If we consider the values in $k = 2$ as a baseline, we can create the following linear equation from there.

$$Frq = B_2 + Dist_K \times Incr_K \quad (1)$$

Using this equation, we can now predict the frequency of each reuse distance where considering B_2 is the base at $k = 2$, $Dist_K$ is the loop bound distance that we want to reach, for example, from 2 to 102 is 100 according to the example code in figure 3, and the $Incr_K$ represents the changes in the frequency for each increment of k, such as 2 is for reuse distance 0, 4 is for reuse distance 1, 3 for 2 and 2 for -1. Table 2 shows the frequency prediction for each reuse distance for $k = 102$, and it exactly matches with the dynamic calculating tool such as PARDA [7].

Reuse Distances	Equation	Pred Frequency
0	5 + 100 * 2	205
1	9 + 100 * 4	409
2	5 + 100 * 3	305
-1	7 + 100 * 2	207

Table 2: Reuse profiles prediction from linear equation.

3.7.2 ***Incorporating Outer Loops and Dilation:*** In this step, we incorporate the single inner loop dilation prediction method into outer loops and observe the changes. If the outer loops bound are kept as the basic 2, increasing the only inner loop bound from 2 to 3 or 4 shows some changes in the reuse distance number itself, which is not seen in the earlier steps. The loop annotated trace for the basic i, j, and k loop variable for figure 1 where all the bounds are set to two looks like the following.
['retval', 'alpha', 'i', '[2', 'i', 'j', '[2', 'j', 'k', '[2', 'k', 'alpha', 'i', 'k', 'A i k', 'i', 'k', 'A i k', 'alpha', 'j', 'k', 'B j k', 'k', 'k', ']', 'k', 'j', 'j', ']', 'j', 'i', 'i', ']', 'i'] If the k bounds are changed from 2 to 3 and 4, we can notice that some reuse distance numbers are constant, such as 0,1,2,3,4,5 and -1 even though the frequencies of them are changing, which is expected. the reuse distance 7 which is present in $k = 2$ loop bound, not present in $k = 3$ or $k = 4$.

K Bounds	Reuse Distance Histogram
2	{0: 35, 1: 11, 2: 37, 3: 25, 4: 5, 5: 12, ***7: 4, 9: 1, 10: 2, 11: 1***, -1: 13}
3	{0: 43, 1: 15, 2: 53, 3: 37, 4: 5, 5: 20, ***9: 6, 12: 1, 13: 2, 14: 2, 15: 1***, -1: 17}
4	{0: 51, 1: 19, 2: 69, 3: 49, 4: 5, 5: 28, ***11: 8, 15: 1, 16: 2, 17: 2, 18: 2, 19:1***, -1: 21}

Table 3: Outer loops shows dilation in reuse distance itself

To address that challenge, we divide the calculations into two distinct parts. The first part involves constant components where the reuse distance remains fixed while only the frequencies change. These can be predicted using the linear equation technique explained in the earlier section. The second part involves scenarios where both the reuse distance and frequency fluctuate. For this part, we have developed a separate tool to predict the varying reuse distances and frequencies. The predictor utilizes three lists to analyze the changing reuse distance numbers. For instance, in Table 3, the three lists are {7, 9, 10, 11}, {9, 12, 13, 14, 15}, and {11, 15, 16, 17, 18, 19}. The predictor examines the relationship between each list, its neighboring lists, and the size dependent on the value of k.

From these three lists and target bound, the predictor constructs equations based on the current list and the neighboring lists, and ultimately generates three types of output: the starting reuse distance for the changing list when $k = n$, the size of the list, and the cumulative list starting from the initial number. Figure 4 shows the input and output for the static predictor.

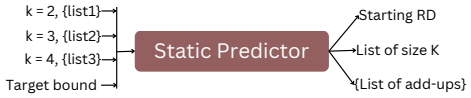

Figure 4: Static predictor for target bound from given lists

For example, with the given reuse distance lists and a value of $k = 102$, the predictor outputs the following: starting number - 207, list size - 104, and a cumulative list starting from the initial number, which is {0, 102, 103, 104, ..., 204}. Using those three outputs, we can now predict the actual reuse distance list for $k = 102$. In the

same way, we can calculate the volatile frequency numbers for the 102 k loop bound.

3.7.3 ***Estimating for Outer Loop:*** At this stage, we aim to estimate the reuse profile for the complete bounds of the nested loops with indices j and k while keeping the outermost loop with index i fixed at a bound of 2. Up to this point, we have computed the reuse profiles for various loop configurations, specifically 2-2-2, 2-2-3, 2-3-2, and 2-3-3. By examining these reuse profiles, we can assess the combined impact of incrementing both j and k by 1, a relationship we define as $Coff_{JK}$. Figure 5 illustrates the overall approach to this task, providing a visual representation of the changes in reuse profiles across these configurations.

Next, we proceed to determine the reuse profile for the innermost loop when its bounds are set to the maximum values, utilizing the technique outlined in the earlier section. In addition, we independently analyze the effect of extending the bounds of j to its maximum while keeping other variables constant. This step is crucial in understanding how changes in j influence the overall reuse profile when j reaches its full bounds.

From the collective analysis of these profiles, we apply Equation 2 to compute the reuse distance frequency, allowing us to quantify how the reuse patterns evolve as the loop bounds are modified. Ultimately, we achieve the full bounds for both j and k while maintaining the bound of i at 2.

$$Frq = B_{22} + Dist_J \times Incr_J + Dist_K \times Incr_K + Coff_{JK} \times Dist_J \times Dist_K \quad (2)$$

In this equation, the frequency of each reuse distance is calculated from B_{22} which is the baseline reuse distance, mainly 2-2-2 is considered as the baseline reuse profile; $Dist_J$ is the distance in loop bound that J is going up; $Incr_J$ is the change in frequency for each J increase and It is also same for the K loop bound in the next term, and finally, $Coff_{JK}$ represents the impact factor of scaling both J and K bounds together.

By following this rule, we can calculate the given bound of I as well and end up at all loop bounds, which is 300-200-102 in this example.

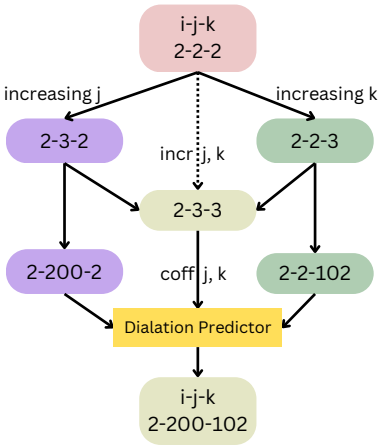

Figure 5: Loop flattening for smaller bound.

Table 4: 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

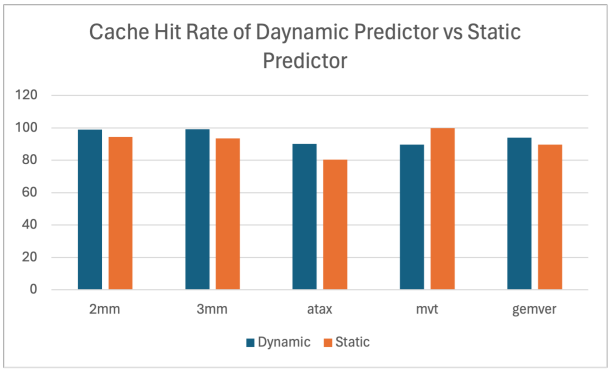

Figure 6: Comparison in Hit Rate between the dynamic and static predictor.

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 by 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 received, and the results are shown in the diagrams. In the X axis, five 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 used a cache with associativities of 20, 20, and 8 and sizes of 16 MB, 25 MB, and 32 KB for the first, second, and third-level caches, respectively, with a cache line of 64 for all three.

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.

Another issue with this technique is that it does not cover the if-else branches inside a loop. It only predicts a larger loop-bound reuse profile from smaller cases. However, if any branch conditions occur, predicting the reuse profile in the static method becomes challenging because branches may not follow the same sequence of calculations.

6 CONCLUSION

Application profiling is significant for performance measurement. Reuse profiles and cache hit rates are critical metrics. This paper presents a unique technique to calculate reuse profiles by gathering information from smaller cases. The approach is independent of loop bounds and is also faster because it follows a static process. Since this method does not require the collection of dynamic traces, it saves time compared to dynamic reuse profile predictors that rely on trace collection. Results show that our tool achieves a cache hit rate of over 95% on average, and in some cases, it even surpasses the dynamic predictor in terms of hits. This technique may be useful when time is more important than compromising a small degree of 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