Cachediff - A tool for localized cache analysis in C/C++ programs

Saimadhav Heblikar*, Sameer Kulkarni*, Viraj Kumar†
*Department of Computer Science and Engineering
PES Institute of Technology,
Bangalore, India
Email: {saimadhavheblikar, ksameersrk}@gmail.com
†Department of Computer Science and Engineering
PES University
Bangalore, India
Email: viraj.kumar@pes.edu

***Abstract*—In this paper, we describe Cachediff, a tool we have built to perform localized cache analysis. Localized cache analysis deals with answering questions about how local code changes between two or more versions of the same C/C++ program affects the cache performance. Previous work on the topic only cover analysis of a program as a whole. We build on their work whilst exploiting certain operating system and compiler features to isolate the local code changes and track their effect on the cache system as whole. We feel that Cachediff will prove useful to instructors, students and professionals to analyze parts of a program for cache performance.**

1. Introduction

Computer Organization and Architecture is a core course in any Computer Science curriculum. The general approach to teaching this course is through use of passive tools like slide shows and diagrams. To make it interesting for the students, the instructor has to come up with active learning tools like animations and simulations. This does indeed require a lot of time and effort from the instructor. Understandably, this does not always happen, and often leads to student losing interest.

Memory management is an important aspect of Computer Organization and Architecture. Like most other topics in this course, it is difficult to learn by conventional procedures. Researchers have come up with cache simulators. Cache simulators are models of the real world. While complex, they are an excellent tool for understanding and verification. However, these cache simulators only help analyze the entire program. They do not provide localized analysis.

Cachediff [1] is a free and open-source tool we have developed to enable students, instructors and professionals alike to perform localized cache analysis. Localized cache analysis here means how changing a small part of a bigger program affects the cache performance of the bigger program.

Students can use it for better appreciation of concepts learnt in their entire Computer Science curriculum. They may use it to verify answers to questions like why row wise traversal is faster than column wise traversal. They can use it to verify that their compiler indeed performs optimization to better cache performance.

Instructors may use it as a teaching aid. They can demonstrate the effect of writing code which is not cache friendly. Instructors may also give assignments and grade them based on which student writes the code which is most cache friendly.

2. Background

Some popular cache simulators are DineroIV [2], CMP $im [3], MSCSim [4] etc.

There are two approaches to cache simulation:

1) Trace driven simulation [5] refers to reading a fixed sequence of trace records representing memory accesses. The entire trace needs to be present on a file before beginning simulation. This requires a lot of space, even for modest size program executions.
2) Execution driven simulation [6] simulates each memory access/ instruction as soon as it happens. Whilst requiring lesser space, it requires more time to process each access/ instruction.

Choosing between the two is a space versus time tradeoff. For this paper, we pick a trace driven simulation approach. However, an execution driven simulation is equally applicable.

To get the memory and instruction traces for a program, we use a tool called Intel Pin® [7]. Pin is a dynamic instrumentation tool. Programs written to make use of the Pin API is called a pintool.

A pintool consist of instrumentation and analysis routines. Instrumentation refers to inserting callbacks to user-defined analysis routines. These callbacks can be inserted anywhere in the program instruction stream. Analysis routines then interact with the feature rich Pin API to extract information about the current status of execution. For example, a user analysis routine can extract information regarding

the current instruction pointer or to count how many times a memory is accessed or an instruction executed.

3. System Description

In this section, we describe the architecture, implementation and work-flow of Cachediff. At a high-level, the input to Cachediff consist of two or more C/C++ programs, along with the respective input file(s), if any. However, for the sake of simplicity, we only consider two input programs.

We first describe certain terms used from here on-wards, which have special meaning with respect to Cachediff.

1) Create an abstract representation of a C/C++ program. This representation is referred to as *File*.
   Creating an abstract representation of *File* involves creating abstract representation of each line of code in a C/C++ program. Each such line of code in a C/C++ program is referred to as a *HighLine*. It is important to remember that a line of code in a C/C++ program is not guaranteed to have a abstract representation. This may be because of the way the program is compiled (For example, a compiler with optimization enabled) or the line may not be doing anything (For example, a blank line).
   Creating an abstract representation of *HighLine* involves creating abstract representation of assembly instructions corresponding to the *HighLine*. Each such assembly instruction is referred to as a *AssemblyLine*.
   We now describe the information contained at each level of abstraction.
   - AssemblyLine: Contains the virtual address of the instruction in the line in the executable.
   - HighLine: This stores the line number of the line and set of *AssemblyLine*'s corresponding to it. It is also aware of the set of virtual addresses corresponding to this line of code in the executable.
   - File: This is essentially a list of *HighLine*'s sorted by line number. We are able to extract a *HighLine* corresponding to a given virtual address.

   To extract the above information, we first create an executable for the C/C++ program. We need to enable debugging information to be embedded into the executable. On the popular and open source GNU GCC [8] compiler collection, the *File* is compiled with the '-g' flag. This information is required to extract details about *AssemblyLine* and *HighLine*.
   Then from executable we need to extract information about each *HighLine*. To do so, we require a tool to dis-assemble the executable. For this purpose, we use objdump [9], which is a part of GNU Binutil. We call it with '-d' to enable dis-assembly and '-l' to label the output with file name and source line number information.
2) The next step is to identify what has changed between the input files. This is the diff or delta corresponding to the two input files. We use the abstract representation created in previous step to compute the diff or delta. This is stored as a tuple. Each element in the tuple corresponds to the list of *HighLine*'s changed in the corresponding *File*. This presents an interesting question - what if the *File*'s have two or more changed blocks? That is, what if there are multiple changes? In such a case, the user may either choose to be warned or choose to ignore. For best result, we suggest that the user limit the number of changed blocks to exactly one. For each *File*, we refer to the block which has changed as the *local diff* block for that *File*.
3) The next step is run the executable. But before that, we need to understand the memory layout of an executable. The executable is divided into many segments. The purpose of a segment could be to consist of code, read only data, shared libraries etc. Each segment has a starting virtual address. When the executable is loaded into memory, the segments are available at that virtual address. If each segment is loaded into the same virtual address every-time, the system is vulnerable against buffer overflow attacks. To overcome this problem, operating systems randomize the memory layout of an executable each time, before load [10]. This process is known as address space layout randomization (ASLR).
   However, ASLR presents a problem to us. If the memory layout of an executable is randomized before every run, every abstract representation we created earlier would be useless. To overcome this, we disable ASLR temporarily. While there are ways to disable ASLR permanently, we caution the reader against it. If the reader wants a permanent solution, they may consider using a virtual machine, to minimize the risks.
4) With ASLR disabled, the executable along with the corresponding input files are run under Intel Pin [7] (using a shared object created for generating the traces). The output is a stream of lines of the format

   ```
   <Instruction pointer> <operation>
   <memory address if applicable>.
   ```

   Let us call this stream *trace*.
   The instruction pointer corresponds to the virtual address of the current *AssemblyLine*. The operation is one of **R**ead, **W**rite or **I**nstruction Fetch. If the instruction involves a memory operand it is specified. If a *AssemblyLine* references more than one memory operand, it is specified in different lines in the *trace*.
5) In this step, we look at a way to locate and isolate parts of the trace which correspond to the *local diff* block.

Define "global trace" as "trace". Define "local trace" as parts of "trace" corresponding to the *local diff* block. Define "non-local trace" as everything in "trace" and not in "local trace".
We now describe our approach to identifying each of the traces described.
The "non-local trace" consist everything from the first line in "global trace" up-to the point where for the first time an instruction from the *local diff* block is executed. Everything from this point on-wards until the last time an instruction from the *local diff* block is executed is ignored. Then consider everything after the point where for the last time an instruction from the *diff block* is executed until the end of the file.
The "non-local trace" consist everything in the "global trace" which is not an instruction belonging to "local trace"
Given the "non-local trace" we identify "local trace" as the difference of "global trace" and "non-local trace"

6) The "global trace" and "local trace" files are fed to a trace driven cache simulator like DineroIV. [2]. The user may also specify the cache architecture of the processor which they want to simulate.
The output of the cache simulator provides many statistics about the cache performance of the program for the given input like L1 miss rate, total number of misses from L2 data cache etc. The user may choose to track all or a subset of these parameters.

4. Evaluation

In this section, we describe our experience using Cachediff [1]. For our experiments, we use a hypothetical processor closely modeled on the lines of INTEL 64® [11]. Table 1 provides details of the values used.

Section 4.1, 4.2, 4.3 and 4.4 make use of the popular 2D-matrix traversal, breadth first traversal, hashing and sorting examples respectively. The parameters we consider for comparison are L1 local misses and L1 global misses. The reason for choosing this parameter is that L1 cache is most crucial to processor performance [12].

4.1. Matrix traversal and multiplication

Commonly used example computer architecture classrooms. There are two ways to traverse a two dimensional matrix Row wise and column wise. Algorithm1 and Algorithm2 describe the algorithms for row wise and column wise traversal. For our experiment, we consider square matrices of size 10×10 to 500×500

From the results in Figure 1, it is obvious that except for very small matrices, row wise matrix traversal outperforms column wise matrix traversal. This verifies our system as most cache systems today use the concept of "spatial locality". Almost all systems also make use of row major representation for storing multidimensional arrays. It is obvious that row major representation better exploits the concept of "spatial locality" than column major representation.

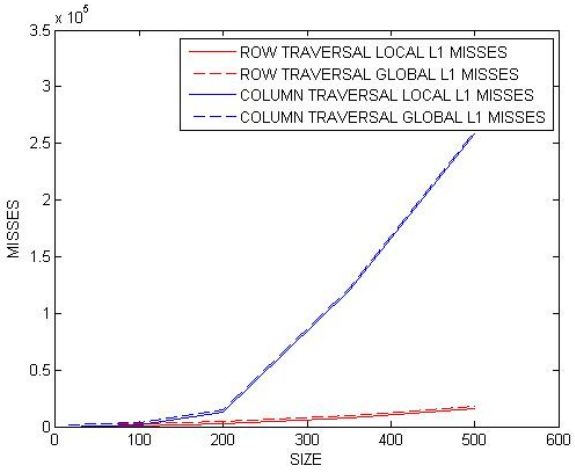
Algorithm 1 Row wise traversal

```
1: for i in 1 to m:
2:     for j in 1 to n:
3:         access element array[i, j]
```

Algorithm 2 Column wise traversal

```
1: for i in 1 to m:
2:     for j in 1 to n:
3:         access element array[j, i]
```

Figure 1. Results of matrix traversal cache simulation



With the superiority of row wise traversal well established, we now shift our attention to the problem of matrix multiplication. The task is to multiply two N$\times$ N matrices. The pseudocode for matrix multiplication is shown below. The order in which the matrix elements are stored and accessed is using row wise traversal. The function *multiply_matrix* will call one of the three matrix multiplication implementations defined in Algorithm3.

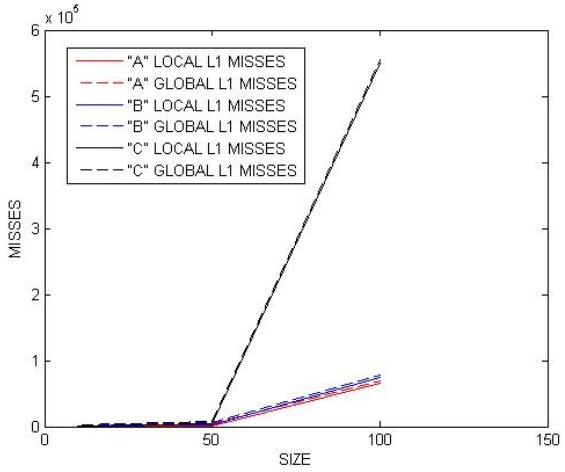
MULTIPLY1 does not make any matrix access in the inner loop. MULTIPLY2 performs row wise traversal in the inner loop. MULTIPLY3 performs column wise traversal in the inner loop. Further explanation for each *multiply_matrix* implementation in Algorithm3 is given in [13]. The result of cache simulation for different matrix multiplication approaches is given in Figure2.

This further verifies Cachediff's ability to isolate and localize cache analysis.

4.2. Adjacency matrix versus Adjacency List

In this section, we try to compare the performance of Adjacency matrix and Adjacency list whilst performing Breadth First Traversal.

Figure 2. Results of matrix multiplication cache simulation



Algorithm 3 Matrix Multiplication Overview

```
1: N ← read_integer()
2: A ← read_matrix(N)
3: B ← read_matrix(N)
4: C ← multiply_matrix(A, B)
5: display(C)
6:
7: procedure MULTIPLY1(A, B)
8:     for i in 1 to n:
9:         for j in 1 to n:
10:            sum ← 0
11:            for k in 1 to n:
12:                sum ← sum + a[i][k] * b[k][j]
13:            c[i][j] ← sum
14:
15: procedure MULTIPLY2(A, B)
16:    for k in 1 to n:
17:        for i in 1 to n:
18:            r ← a[i][k]
19:            for j in 1 to n:
20:                c[i][j] ← c[i][j] + r * b[k][j]
21:
22: procedure MULTIPLY3(A, B)
23:    for j in 1 to n:
24:        for k in 1 to n:
25:            r ← b[k][j]
26:            for j in 1 to n:
27:                c[i][j] ← c[i][j] + a[i][k] * r
```

We now describe the methodology used to perform the analysis. The density D of a graph is given by

$$D = \frac{|E|}{|V| * (|V| - 1)}$$

where $|E|$ is the number of edges in the graph and $|V|$ is the number of vertices. We take the liberty to define sparse, median and dense graphs as graphs with $D = 0.1$, $D = 0.5$ and $D = 0.9$ respectively. The same graph is represented as adjacency matrix and adjacency list.

For each graph density defined above, we created N graphs[1]. Each of these N graphs were converted to Adjacency Matrix and Adjacency List representations. On each such graph, we perform Breadth First Traversal on every forest in the graph for both representations.

The results of our runs is given in Table 2.

We conclude the following by a cursory look at the results.

- Adjacency Matrix performance is same irrespective of the density of the graph.
- The performance of Adjacency List improves as the density of the graph increases.
- For graphs of the same size and when size is known beforehand, Adjacency Matrix always outperforms Adjacency List in terms of cache performance.

Our conclusions from the results are in coherence with existing literature dealing with analysis of cache performance of graph algorithms [15].

4.3. Hashing – Linear Probing vs Seperate Chaining

Linear Probing and Seperate Chaining are two approaches to hashing. For our experiment we consider insert and search (both successful and unsuccessful) operations.

We fix the hash table size as 1000 to reduce the number of variables in the model. We then perform N insertions and S searches on the hash table. The results of simulation can be seen in Figure3. To avoid clutter, we only consider the local L1 misses in Figure3. For local analysis in the search operation, using Cachediff, we are able to ignore the insertion operation which was done before searching. Operation count on x-axis refers to number of insertions or searches, as applicable.

We see that Linear Probing performs better than Seperate Chaining. This is because Linear Probing exploits "spatial locality" better than Seperate Chaining. In Linear Probing, the elements are bound to be in the hash table, whilst in Seperate Chaining the elements may be anywere in the memory. This is inspite of the fact that Seperate Chaining is traditionally better than Linear Probing [15]. To not divert from Cachediff and localized cache analysis, We also do not consider re-sizing the hash table.

4.4. Quick Sort versus Bubble Sort

Everybody knows how awful bubble sort is! [16]. Yet, it is one of the first sorting algorithms taught to students.

1. The graphs were created using Wolfram Mathematica [14]. For sake of reproducibility, we have included the code used to generate the graph in Cachediff [1] source tree.

Figure 3. Results of hasing cache simulation

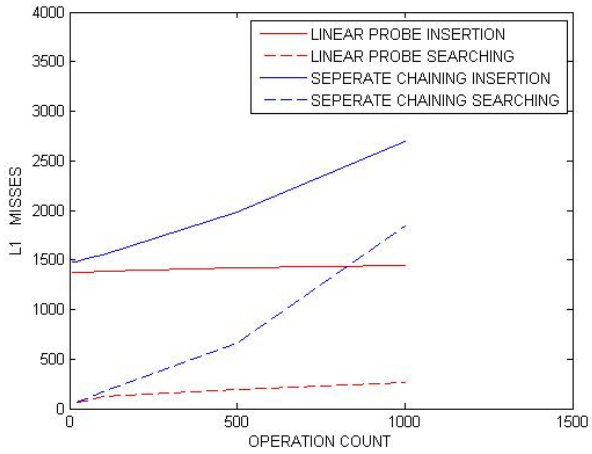


TABLE 1. DETAILS OF HYPOTHETICAL PROCESSOR

Description	Value
l1 instruction cache size	32k
l1 data cache size	32k
l1 instruction block size	64
l1 data block size	64
l1 instruction associativity	8
l1 data associativity	8
l2 unified cache size	256k
l2 unified block size	64
l2 unified associativity	8
l3 unified cache size	8M
l3 unified block size	64
l3 unified associativity	8
l1 data write policy	write back

On the other hand, one of the best algorithms for sorting is the Quick Sort. It is used in some form or another in popular languages' sort function implementation. A complaint against quick sort is its branch prediction abuse [17].

While Quick Sort belongs to a better complexity class $\mathcal{O}(n \log n)$as compared to Bubble Sort$\mathcal{O}(n^2)$, our empirical results show that Bubble Sort competes well with Quick Sort when it comes to cache performance in terms of L1 misses. But we make it a point to mention that the number fetches requested by Quick Sort still remains $\mathcal{O}(n \log n)$, whilst that of Bubble Sort remains $\mathcal{O}(n^2)$

TABLE 2. RESULTS FOR BREADTH FIRST TRAVERSAL CACHE SIMULATION

Type	Density	Local L1 Misses	Global L1 Misses
Adjacency Matrix	dense	8	3015
Adjacency List	dense	75	9173
Adjacency Matrix	median	8	3017
Adjacency List	median	59	6461
Adjacency Matrix	sparse	8	3018
Adjacency List	sparse	26	3712

TABLE 3. RESULTS OF SORTING CACHE SIMULATION

Algorithm	Size	Local L1 Misses	Global L1 Misses
Bubble Sort	10	30	1849
Quick Sort	10	30	1849
Bubble Sort	100	25	1864
Quick Sort	100	25	1864
Bubble Sort	500	68	2032
Quick Sort	500	68	2032
Bubble Sort	1000	125	2088
Quick Sort	1000	138	2090

5. Conclusion and Future Work

The result obtained using our localized cache analysis is coherent with verified result obtained elsewhere. The results are also obtained at near execution speeds plus a very small overhead. We feel that the overhead is acceptable in a classroom environment. We feel that this paper illustrates the concept of localized cache analysis very well.

We have used common classroom examples to show that Cachediff can be used as a teaching aid for the Computer Organization and Architecture class.

For the future,we plan to create plugins for Integrated Development Environments (IDE) like Eclipse or NetBeans. This way, students and instructors would be able to graphically select or view the *diff block*.

We will also wish to maintain a regulary updated online repository which would contain information about different processors in a format suitable for Cachediff. This can be used to compare peformance of the same or different programs on different processors.

We will also test Cachediff for production and industrial workloads. This would make it useful to professionals. We would also like to explore how we can make Cachediff support multi-threaded and/or multiprocessor environments.

References

[1] S. Heblikar, S. Kulkarni, and V. Kumar. Cachediff - localized cache analysis for c/c++ programs. [Online]. Available: https://github.com/sahutd/cachediff

[2] J. Edler and M. D. Hill, "Dinero iv trace-driven uniprocessor cache simulator," 1998.

[3] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp $im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36.

[4] L. Coutinho, J. L. D. Mendes, and C. A. Martins, "Mscsim-multilevel and split cache simulator," in *Frontiers in Education Conference, 36th Annual*. IEEE, 2006, pp. 7–12.

[5] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys (CSUR)*, vol. 29, no. 2, pp. 128–170, 1997.

[6] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1105734.1105747

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[8] (2015, nov) Gcc, the gnu compiler collection. [Online]. Available: https://gcc.gnu.org/

[9] (2015, nov) objdump(1) - linux man page. [Online]. Available: http://linux.die.net/man/1/objdump

[10] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[11] Intel 64 and ia-32 architectures optimization reference manual. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[12] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[13] G. Robboy. Writing cache-friendly code. [Online]. Available: http://web.cecs.pdx.edu/ jrb/cs201/lectures/cache.friendly.code.pdf

[14] Wolfram Research, Inc., "Mathematica." [Online]. Available: https://www.wolfram.com

[15] J. R. Black, C. U. Martel, and H. Qi, "Graph and hashing algorithms for modern architectures: Design and performance." in *Algorithm Engineering*. Citeseer, 1998, pp. 37–48.

[16] O. Astrachan, "Bubble sort: An archaeological algorithmic analysis," *SIGCSE Bull.*, vol. 35, no. 1, pp. 1–5, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/792548.611918

[17] K. Kaligosi and P. Sanders, "How branch mispredictions affect quicksort," in *Algorithms–ESA 2006*. Springer, 2006, pp. 780–791.