Shiv Nadar University
Department of Computer Science and Engineering

CSD208: Computer Architecture

Project

Cache Mispattern and Mispredictibility Analysis

Illisha Singh (1710110146)
R Deepa (1710110263)

Abstract

Improving cache performance requires understanding cache behaviour. The efficiency of a computer system in executing a program is highly dependent on the effectiveness of the memory hierarchy to supply requested data quickly. Unfortunately, a program's algorithm memory access pattern can be quite complicated and there is no way to analyse arbitrary access patterns. Accordingly, we model algorithms as a combination of simple access patterns whose cache performance characteristics can be quantified. I This paper describes a model for studying the cache performance of algorithms in a direct-mapped cache. Using this model, we analyse the cache performance of several commonly occurring memory access patterns: (i) sequential and random memory traversals, (ii) systems of random accesses, and (iii) combinations of each. In this paper, we also examine the dependence of the cache miss rate on cache size through simulation. This paper also presents an interactive visualization tool that uses a three-dimensional plot to show miss rate changes across program data sizes and cache sizes and its use in evaluating compiler transformations. Other uses of this visualization tool include assisting machine and benchmark-set design. The tool can be accessed on the Web at http://www.cs.rochester.edu/research/locality.

1. INTRODUCTION

This paper analyses cache miss pattern behaviour and its prediction using a few algorithms; presents a simulation of cache miss rate for set associative caches on different workloads; gives an introduction of a visualisation tool that shows a three-dimensional plot of cache miss rates and then finally concludes by giving various uses of cache miss rate prediction.

The second section talks about why we need to analyze and predict cache miss pattern and miss rates. The third section proposes and investigates an arena for concrete analysis of algorithms. We call this new area cache performance analysis of algorithms because we are interested in approximating as closely as possible the number of cache misses an algorithm incurs. In modern processors cache misses are very costly steps, in some cases costing 50 processor cycles or more In the tradition of concrete analysis of algorithms, cache misses are genuinely costly, and it is appropriate to analyse them. The key application of cache performance analysis is towards the cache conscious design of data structures and algorithms. The goal of the work in this paper is to assemble a useful set of analysis tools for understanding the cache performance of algorithms.

We consider two types of traversals: a simple scan traversal that traverses memory sequentially and a permutation traversal that visits every block in the contiguous segment a fixed number of times. Both traversals are common patterns in algorithms. For permutation traversals, we give a simple expression for the expected number of cache misses per access in theorem 1 assuming the permutation order is chosen uniformly at random. In practice, the expression is a good predictor of the cache performance of an arbitrary permutation traversal.

A system of random accesses is defined by a set of random processes that access memory in a stochastic, memoryless fashion. In this paper we analyze systems of random accesses running for a finite number of steps. We apply the analysis to obtain theorem 2, which describes the cache performance of such a system interacting with a scan traversal. Quantifying the interaction between the two access patterns is important, as it yields better predictions than that obtained from analysing each access pattern independently.

We assume that items are only brought into the cache when they are accessed and are only removed from the cache when they are evicted by another conflicting access. This paper focuses on direct mapped caches, a cache design commonly found in modern machines. In direct-mapped caches, the data item brought into the cache can only go to one place. The result is that our analyses are both different and likely more useful in practice than cache analyses for fully associative caches. There have been several studies that have tried to quantify the cache performance of programs by summarizing or analysing actual memory access traces. The work in this paper differs from this in that we do not examine the trace of a program, but just, the actual algorithm itself.

The fourth section gives a simulation of the cache miss rates. Most modern processor systems include a cache hierarchy, and will continue to do so in future. There are numerous papers discussing the use of analytical cache models to predict cache behaviour. It is clear that if a workload is small and fully fits within the size of a particular cache, the miss rate will be small. However, if the workload is large, the cache miss rate is observed to decrease as a power law of the cache size, where the power is approximately -1/2. This is the well known $\sqrt{2}$ rule, where if the cache size is doubled, the miss rate drops by the factor $\sqrt{2}$. This section gives simulations which test these theoretical ideas. It is found that the dependence of miss rate on cache size arises from the time dependence of the cache reference pattern exhibited by a particular workload. In particular, the root(2) rule emerges because most large complex workloads display a cache re-reference timing pattern, which obeys a power law. The power law, observed for the cache references, shows exponents ranging from -1.7¡β¡-1.3. The corresponding power law for the miss rate, the square root rule, ranges from -0.7¡α¡-0.3.

The fifth section demonstrates the visualization tool in compiler evaluation. It measures the effect of a program transformation not just for one program execution on a single machine but for all program inputs for all cache configurations. In addition, the sixth section discusses other uses of this tool in cost effective cache design and benchmark set design and the various uses of cache miss rate prediction.

2. NEED FOR PATTERN PREDICTION

The memory system is a major performance and power bottleneck in embedded systems, especially for multimedia applications. Most multimedia applications access a stream type of data structures with regular access patterns. It is observed that conventional caches behave poorly for the stream-type data structure. Therefore, prediction-based pre-fetching techniques have been extensively researched to exploit regular access patterns. Pre-fetching, however, may pollute the cache if the prediction is not accurate and needs extra hardware prediction logic. To overcome these problems, we propose a novel hardware pre-fetching technique that is assisted by a static analysis of data access pattern with stream caches. With the proposed stream cache architecture, we could achieve significant performance improvement compared with the conventional cache architecture. General cache schemes such as direct and set-associative caches can attenuate the gap on average, there is still much room for optimization if the memory access pattern is known a priori.

2 MODES OF CACHE PREDICTION:

Several kinds of research have focused on prefetching techniques to exploit the stream-like access pattern. Prefetching can be triggered either by a hardware mechanism, or by a software instruction, or by a combination of both.

1. HARDWARE MECHANISM: The hardware approach detects accesses with regular patterns and speculatively issues prefetches at run time, which may cause some run-time overhead and a cache pollution problem in case of misprediction. Software Mechanism: The software approach relies on the compiler to analyze the program and to insert prefetching instructions, which may increase the code size.

2. SOFTWARE MECHANISM: The software approach relies on the compiler to analyze the program and to insert pre-fetching instructions, which may increase the code size.

It is found that the dependence of miss rate on cache size arises from the time dependence of the cache reference pattern exhibited by a particular workload. In particular, the $\sqrt{2}$ rule, where if the cache size is doubled, the miss rate drops by the factor $\sqrt{2}$, emerges because most large complex workloads display a cache re-reference timing pattern, which obeys a power law.

We analyse the cache performance of several commonly occurring memory access patterns:

- sequential and random memory traversals
- systems of random accesses, and
- combinations of each.

In modern processors, cache misses are very costly steps, in some cases costing 50 processor cycles or more. In the tradition of concrete analysis of algorithms, cache misses are genuinely costly, and it is appropriate to analyse them. From the cache's perspective, an executing algorithm is simply a sequence of memory accesses. Unfortunately, an algorithm's memory access pattern can be quite complicated and there is no way to analyse arbitrary access patterns. Accordingly, we model algorithms as a combination of simple access patterns whose cache performance characteristics can be quantified. We, thus, focus on two basic access patterns, a traversal of a contiguous segment of memory and a system of random accesses.

We assume there is a large memory that is divided up into M blocks and a smaller cache that is divided up into C blocks. Memory is indexed 0 to M - 1 and the cache is indexed 0 to C - 1. Although C $\leq$ M, we assume that C is fairly large, at least in the hundreds and more likely in the tens of thousands. We assume the simple mapping where memory block I is mapped to cache block z mod C. At any moment of time each block y in the cache is associated with exactly one block of memory z such that y = x % C. In this case, we say that block 3: of memory is in the cache. Initially, none of the blocks to be accessed are in the cache. In reality, an algorithm reads and writes to variables where a variable is stored as part of a block or as multiple blocks. A read or write to a variable that is part of a block is modelled as one access to the block. A read or write to a variable that occupies multiple blocks is modelled as a sequence of accesses to the blocks. A write to cache block y = x % C is written to memory block x when a miss occurs, that is, when block t is accessed where Y = z % C and z $\neq$ x. A write buffer allows the writes to location x to propagate to memory asynchronously with high probability. **The cache performance of an algorithm is measured by the number of misses it incurs**. Naturally, an algorithms overall performance is not just a function of the number of misses, but also a function of the number of memory accesses, the runtime penalty incurred due to each cache miss, and the cost of other operations.

Mode of analysis 1: Traversal
A traversal with block access rate K accesses each block of a contiguous array of N/K blocks exactly K times each. Hence, there are a total of N accesses in a traversal.

Scan Traversal: A scan traversal starts with the first block, accesses it K times, then goes to the next block accessing it K times, and so on to the last block. Scan traversals are extremely common in algorithms that manipulate arrays. If B array elements fit in a block then a left-to right traversal of the array is a scan traversal with block access rate B.

Theorem 1. *A scan traversal with block access rate K has* $1/K$ *cache misses per access.*

Permutation Traversal: A permutation traversal starts by accessing a memory block chosen uniformly at random. At any point in the permutation

traversal, if there are k accesses remaining and memory block I have j accesses remaining, then memory block x is chosen for the next access with probability j/k.
A traversal of the linked list is essentially a permutation traversal with block access rate B; where the nodes are allocated randomly and B nodes fill a block.

Theorem 2. *Assuming all permutations are equally likely, a permutation traversal with block access rate K of N/K contiguous memory blocks has l/K misses per access if* $N \leq C$ *and*

$$1 - \frac{(K-1)C}{N}$$

expected cache misses per access if $N \geq KC$.

To illustrate our result for permutation traversals, we apply it to predicting the cache performance of a preorder traversal of a random binary search tree. The binary search tree implementation we consider is pointer-based. Each node of the tree consists of a key value, a data pointer, and pointers to its left and right children. The keys of the nodes, hence the structure of the tree, are determined independently of the node allocation order. In a recursive implementation of a preorder traversal, we start at the root node, examine its key, then recursively visit the nodes of the left subtree, and then the nodes of the right subtree. Assume that exactly L tree nodes fit in a cache block. A preorder traversal, then, is a permutation traversal with K = 3L.

We model the accesses to the keys as a permutation traversal with K = L, and the remaining accesses to the child pointers as hits. If n is the number of tree nodes and C is the cache size in blocks, theorem above gives the total number of misses to be n / L when n < CL and n − (L − 1)C otherwise. This is because when the key of a node is visited, the next access will always be to the left child pointer. Furthermore, the right child pointer will be accessed next for the majority of nodes (the leaves) or may be accessed soon after.

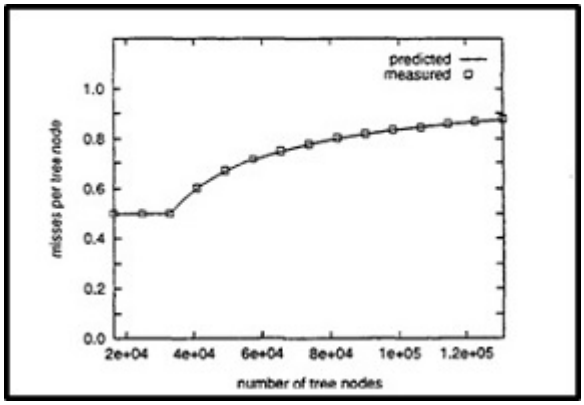

Figure 1: Comparison of the measured cache performance of a preorder tree traversal with that predicted by our model.

In this implementation, a tree node consisted of a key, data pointer, and left and right child pointers, totalling 32 bytes. The simulated cache was a megabyte in size with 64-byte blocks, so L=2 and C=214= 26,384. Figure 1 shows the misses per tree node measured for a preorder traversal of a randomly generated tree for varying sizes n.

Mode of analysis 2: Random Access

In a random access pattern each block x of memory is accessed statistically, that is, on a given access block x is accessed with some probability. Often a random access pattern can be analyzed approximately by assuming the independent reference assumption. In this model, each access is independent of all previous accesses. In a system of random accesses, the cache is partitioned into a set R of regions and the accesses are partitioned into a set P of processes. In practice, the processes are used to model accesses to different portions of memory that map to the same portion of the cache

Let λ_{ij} be the probability that region i$\in$ R is accessed by process j $\in$ P. Let r_{i} be the size of region i in blocks, so that C = Σr_{i}, $i \in R$ Furthermore, let λ_i be the probability that region i is accessed, that is, $\lambda_i = \Sigma r_{\mathrm{ij}}$ j$\in$ P.

Theorem 3. *In a system of random accesses, in the limit as the number of accesses goes to infinity, the expected number of misses per access is*

$$1 - \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} {\lambda_{ij}}^2$$

It is helpful to define the quantities

$$\eta_i = \frac{1}{\lambda_i} \sum_{j \in P} {\lambda_{ij}}^2$$

the probability that an access is a hit in region i, and

$$\eta = \sum_{i \in R} \eta_i$$

the probability that an access is a hit. The probability that an access is a miss is simply $1 - \eta$.

To illustrate the application of collective analysis we consider the cache performance of a simple algorithm that can be modelled as random access. The algorithm uses two arrays: one that is the size of the cache, and the other one-third the size of the cache. The algorithm loops forever where on each iteration it chooses two random elements of the first array and stores their sum in a randomly chosen element of the second array. To apply collective analysis to this algorithm we divide the cache into two regions: Region 0 is one third the size of the cache and holds blocks from the first one-third of the first array and all of the second array; Region 1 is two-thirds the size of the cache and holds the remaining blocks of the first array. We model the accesses to the arrays by

two processes: Process 0 performs the reads from the first array and process 1 performs the writes to the second array. Note that process 0 accesses memory twice as frequently as process 1. The per-region access parameters are :

$$\lambda_{00} = \frac{2}{9}, \lambda_{01} = \frac{1}{3}, \lambda_{10} = \frac{4}{9}, \lambda_{11} = 0, \lambda_0 = \frac{5}{9}, \lambda_0 = \frac{4}{9}$$

Note that this model is an approximation to the actual access pattern of the algorithm as the read and write process accesses are deterministically interleaved- we know that every third access is to the second array, for example. However, this model can yield a good approximation for the hit rate. the predicted number of hits per access for this example are

$$\eta = \frac{9}{5}\left(\left(\frac{2}{9}\right)^2 + \left(\frac{1}{3}\right)^2\right) + \frac{9}{5}\left(\frac{4}{9}\right)^2 = \frac{11}{15}$$

This yields an expected miss ratio of about 26.7%, which is within .2% of the miss ratio measured from an implementation of this algorithm using the above methodology.

We analyzed several common memory access patterns including traversals, random accesses, and traversals with random accesses combined. We experimentally validated the analytic formulas using trace-driven cache simulation. Naturally, there are many other memory access patterns that arise in algorithms, and new techniques need to be devised to analyze them. Extending this work to determine the cache performance of other memory access patterns will hopefully aid algorithm implementors in understanding the impact of a design choice, primarily in cases where memory access performance is an issue. An interesting challenge in extending this work is in determining the number of caches misses that occur when two or more access patterns occur in conjunction, that is when the accesses from each pattern are interleaved in some way. Unlike traditional operation count analysis, the number of cache misses incurred by two interleaved access patterns is not necessarily the sum of their individual miss counts. The analysis of a scan traversal combined with random access is one example of this, and only full analysis allowed us to understand which additional misses caused by their interaction were a large contribution and which were not.

The above study focuses on direct-mapped caches that fetch data upon access, however, there are other kinds of caches that are used in modern architectures. Some processors support prefetching and nonblocking loads. With these features, a data item might be brought into the cache before it is needed, essentially reducing the number of misses incurred by an algorithm. To achieve this, prefetching often requires **reference prediction** by either the hardware or software. For scan traversals, this is straightforward, but it can be difficult for the random access patterns studied here. However, further study and understanding of the cache performance of memory access patterns could benefit prefetching analyses. Some caches are k-way set-associative, where a given block

of memory can map to any one of k blocks in the cache. While the cache performance of traversals and uniformly random access patterns do not benefit from set-associativity, set associativity tends to reduce the cache misses per access for other access patterns and in practice.

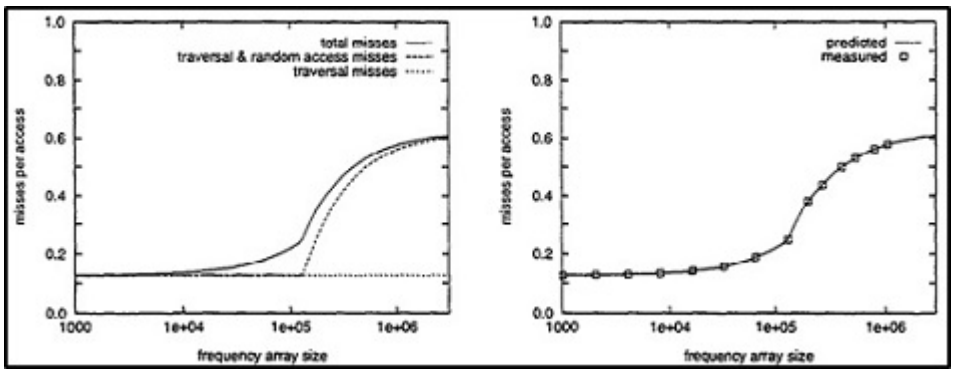

Figure 2: The cache performance of frequency counting. Left: a graph of the predicted misses per access showing the contributions of the real data Right: a comparison of the predicted performance with that measured from simulation.

4. SIMULATION OF DEPENDENCE OF CACHE MISS RATE ON CACHE SIZE

In order to explore the dependence of cache miss rate on the cache size, a proprietary simulator [16] has been used. As input the simulator uses design parameters that describe the organization of the processor and a trace tape. It produces a very flexible cycle accurate model of the processor. With this tool we are able to model two levels of cache hierarchy leading to main memory, numerous pipeline designs, various issue width superscalar designs, in-order execution processing and out-of-order execution processing. This tool has mainly been used for work on IBM zSeries processors.

We focus on the behavior of large complex workloads, which adequately stress the memory structure. The workloads used were largely on-line transaction processing (OLTP) and a processor simulator. The workloads have been divided up into 4 categories: legacy workloads, modern workloads, SPEC Int workloads and floating point workloads. The legacy workloads are large server workloads, mainly written in assembly language. The modern workloads are also large server workloads, but written in either C++ or Java. The floating point workloads tend to be scientific calculations, some of which are from the SPEC suite, and some from other sources.

We present an analysis of cache miss rate for set associative caches, through simulation on the above mentioned workload.

Set Associative Caches The reference pattern is unchanged than in the case of direct mapped cache, but the probability of finding the referenced line in the cache is modified. It is modified because s separate references to the same

cache congruence class need to occur prior to the particular re-reference that causes a cache miss, if s is the number of sets (ways) in the cache, with each subsequent new cache reference the entry in question is pushed into a new set in the cache, until it is finally ejected from the cache.

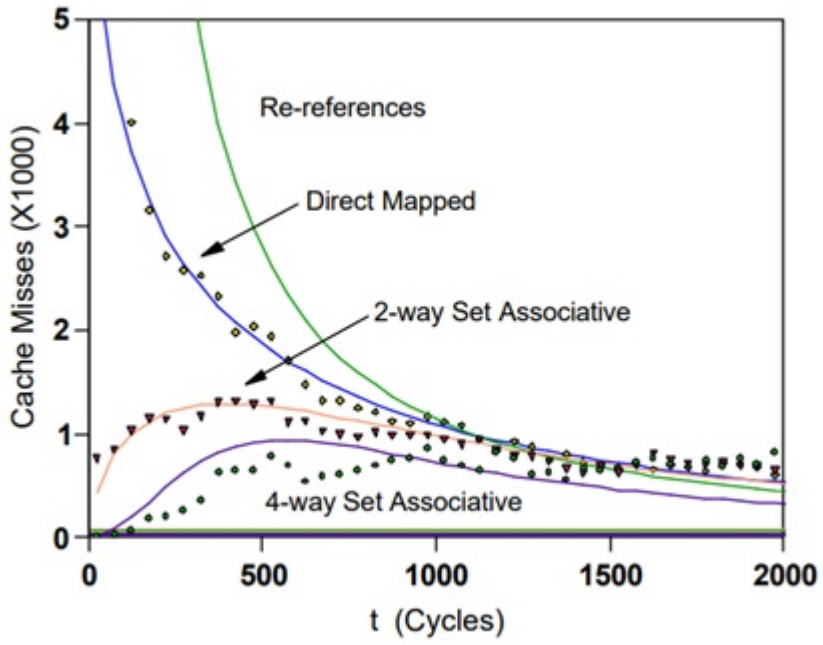

Figure 3: The time dependence for cache misses for set associative caches.

This analysis captures the essential features of the time dependent miss curve for a set associative cache. In the figure data from one workload is shown for 4 kB caches, which are direct mapped, 2-way set associative and 4-way set associative. In addition the theoretical curves for the re-reference pattern and the miss patterns for each of the caches is also expressed.

It is clear that our analysis accurately accounts for the data. By increasing the set associativity one does a better job of retaining useful information in the cache and the area between the curves, associated with cache hits, expands. There is another way of qualitatively looking at the shape of these curves for the set associative case. Before a cache miss may occur, one must wait for some delay time, t0 , before the cache entry is in the last set (LRU set) of the cache. At this point the problem looks like the problem for a direct mapped cache. Qualitatively, one would expect a curve shaped like the direct mapped case, but offset by this average delay time. This essentially leads to zero probability of a miss for times shorter than t0 . Now in a real system this delay time will vary for each cache congruence class and for each occurrence of a cache miss.

Therefore, the average behavior will smooth off the abrupt onset of the curves. This causes the type of peaked structure observed.

5. CACHE MISS RATE VISUALIZATION AND TOOLS:

Two-dimensional graphs cannot show miss rates for all program inputs and cache configurations at the same time. This section presents a Web-based tool that displays miss rates as a three-dimensional plot. We use this tool to evaluate a compiler and discuss other uses in machine and benchmark-set design.

Visualization Tools

The interactive tool displays the miss rates of a program across program inputs and cache configurations. By varying the cache size (x-axis) and the data size (y-axis), the tool generates a 3D graph of the miss rates covering a broad region of interest. The miss rate surface is formed by connecting every three adjacent points (along the x and y directions) to approximate, via linear interpolation, the miss rates over the region. While a 3D graphing aids in quick exploration over a large combination of cache and data sizes, two-dimensional graphs can also be generated for a specific cache or data set size. Appendix I, which can be found on the Computer Society Digital Library at http:// computer.org/tc/archives.htm, gives examples of both 3D and 2D outputs of our tool as well as a detailed description of the tool interface.

The visualization tool can be accessed via the Internet at http://www.cs.rochester.edu/research/locality with any browser equipped with Java 1.4 JVM and Java 3D. The system currently runs on practically all x86 machines (Windows, Linux), Sun, SGI, and IBM platforms.

6. USES OF MISS RATE PREDICTION

1. COMPILER EVALUATION:
   Typically, compiler transformations are evaluated by testing programs for a few inputs on a few machines. While running on real systems allows accurate measurement of a complete system, the few results cannot reliably predict the effect on other program inputs and on other machines. By using miss-rate prediction, a compiler writer can examine memory behavior across all program inputs and all cache sizes. The visualization tool is a convenient method for quickly summarizing the results to see if a certain compiler transformation will provide better performance on a certain memory hierarchy.

   Although the tool helps to evaluate a compiler transformation, the average miss rate itself is not specific enough to drive compiler transformations. Still, similar prediction techniques can be used at a finer granularity. Fang

et al. found an accurate miss-rate prediction for most memory references in SPEC2Kfp programs. Such prediction is directly useful to a compiler.

2. COST EFFECTIVE MEMORY HIERARCHY DESIGN:
   Today's computing centers often use thousands of processors to run a few large applications. Rather than having to simulate numerous inputs and extrapolating how the system would perform from those runs, designers could save both time and money by being able to see the change in miss rates as cache size changes. Additional uses include:

   - Allowing customers to quickly determine a balance between price and performance for memory configurations
   - Evaluating the cache performance of data sets too large to simulate on any existing machine
   - Providing insight into existing systems used to execute new or larger applications
   - Determining whether a new system upgrade can provide improved use of the memory hierarchy.

3. BENCHMARK SET DESIGN:
   With accurate estimates of cache performance across numerous inputs, benchmark design may be improved to allow faster evaluation of systems and optimizations. By building benchmarks that run on values directly after knees in the data size versus miss rate plots, the smallest possible data size for a given miss rate could be used in testing. For two SPEC programs, Table 2 shows the smallest data input size (labeled mini-ref) that gives a similar cache miss rate for 64 KB and 1 MB cache as the reference input does. We obtained the new input size from the miss-rate prediction from Fig. 7 without testing any additional inputs. The new input is a factor of 4 or 12 smaller, but its miss rate differs by no more than 0.6 percent from the larger input. A smaller input saves time in testing. The right input size depends on the program as well as the cache configuration. The prediction tool helps a user to quickly find the smallest input size that yields a particular cache miss rate. The miss-rate prediction can ensure that a benchmark set includes all miss rates and the smallest program runs for these miss rates. Furthermore, the suggestion is parameterized and, therefore, tailored to any cache configuration being considered.

7. CONCLUSION

We analysed several common memory access patterns including traversals, random accesses, and traversals with random accesses combined. We experimentally validated the analytic formulas using trace-driven cache simulation. Naturally, there are many other memory access patterns that arise in algorithms, and new techniques need to be devised to analyse them. Extending this work to

determine the cache performance of other memory access patterns will hopefully aid algorithm implementers in understanding the impact of a design choice, primarily in cases where memory access performance is an issue. An interesting challenge in extending this work is in determining the number of caches misses that occur when two or more access patterns occur in conjunction, that is when the accesses from each pattern are interleaved in some way. Unlike traditional operation count analysis, the number of cache misses incurred by two interleaved access patterns is not necessarily the sum of their individual miss counts. The analysis of a scan traversal combined with random access is one example of this, and only full analysis allowed us to understand which additional misses caused by their interaction were a large contribution and which were not.

8. REFERENCES

1. E.Ladnert, Richard, et al. "Cache Performance Analysis of Traversals and Random Accesses*" Homes.cs.washington.edu, NSR Grant CCR-9732828, homes.cs.washington.edu/ lamarca/pubs/ladner-fix-lamarca-soda99.pdf. Department of computer science and Engineering, Washington, Seattle, WA 98195 OXerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304

2. Kartik Nagar and Y N Srikant. 2016. Refining cache behavior prediction using cache miss paths. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January 2017), 25 pages. DOI: http://dx.doi.org/10.1145/0000000.0000000

3. Zhong, Yutao and Dropsho, Steven and Shen, Xipeng and Studer, Ahren and Ding, Chen. (2007). Miss Rate Prediction Across Program Inputs and Cache Configurations. Computers, IEEE Transactions on. 56. 328-343. 10.1109/TC.2007.50.

4. Hartstein, A., et al. "(PDF) On the Nature of Cache Miss Behavior: Is It $\sqrt{2}$" Https://Www.semanticscholar.org/, IBM – T. J. Watson Research Center PO Box 218 Yorktown Heights, NY 10598 USA, $www.researchgate.net/publication/229004611_On_the_Nature_of_Cache_Miss_Behavior_Is_It_2$.