A Mathematical Cache Miss Analysis for Pointer Data Structures

Hongli Zhang and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{hongliz,martonosi}@ee.princeton.edu

Abstract

As the gap between processor and memory performance widens, careful analyses and optimizations of cache memory behavior become increasingly important. While analysis of regular loop-based scientific programs has made significant progress, the analysis of less regular reference patterns has lagged somewhat. This paper presents basic analytic expressions for cache miss behavior in fundamental types of irregular programs: linked list traversals and binary tree searches. We compare our analytic expressions to simulation-based results. Although quite elementary, these simple expressions will form the foundation for our planned work in analyzing the behavior of database and other applications which have linked lists, trees, and other pointer data structures at their core.

1 Introduction

As the speed gap between processor and memory widens, the study of cache memory is increasingly important. Through the effort of many researchers, formal theory for cache behavior of arrays in numeric programs have become well-established; numerous code and memory layout transformation methods have been proposed and implemented [12, 6].

For pointer data structures, however, the success has been limited. These structures are dynamically allocated at run time, and their irregular data placement and pointer chasing characteristics pose a big barrier for formalized methods. Nonetheless, some sub-problems, such as hardware and software prefetching [11, 15], have begun to show success [11, 15]. Other efforts have examined making malloc and garbage collectors cache-conscious [2, 4], reorganizing data layout, or even transforming code dynamically to change data access order to optimize program cache behavior [5, 3].

This study is a first step towards extending CMEs [6] to pointer-based programs. The goal is to obtain some formal mathematical method to characterize the cache behavior of pointer data structures (PDSs). An analytical model of cache misses for PDS will ultimately help us devise general and formal means to optimize PDS cache behavior.

We studied typical PDS linked lists and binary trees, getting analytical representations for the number cache misses. We then validate these analytic expressions with simulations. For linked lists, over 96% of the cache misses are heap misses, which are cold and conflict misses incurred by the list nodes themselves and are determined by

the number and size of list node. For recursive binary trees, over 93% of the misses are heap misses. Since linked lists and binary trees are two very common PDS types, a deep understanding and mathematical characterization of their cache behavior will help us to model more complicated PDSs, such as B-Trees later on.

Later sections of this paper are organized as below: Section 2 discusses the gnu malloc routine, which serves as background knowledge for following analysis. In Section 3, we present our detailed cache miss analysis and validation for linked list and binary tree. Section 5 extends our analysis from contiguous allocations to cases in which the allocated nodes are only piecewise contiguous. Finally, Section 6 discusses related work and Section 7 concludes this paper.

2 Gnu Malloc Routine

We discuss here the details of the memory allocation our analysis and measurements are based on. Gun malloc uses a variant hybrid first-fit/segregated algorithm [7]. When the requested size is bigger than half of a page size, the returned block is retrieved from a doubly linked free list using a first-fit strategy. If the first-fit free block is bigger than requested, the remaining left-over is re-linked into the free list. When the requested size is smaller than half of a page size, it is rounded to the nearest power of 2 and retrieved from the segregated free fragment list with the corresponding power of 2. If the correct-sized list is empty, a new page is requested. The first fragment is returned, while the other ones are added into the front of list one by one. The resulting list is in descending order by addresses.

Consider a PDS with nodes whose size is less than half of a page size. The PDS is built with each of its nodes allocated one by one. If there are no intervening allocations of another similar-sized PDS nodes from the same list, the resulting memory layout will be generally contiguous. The main differences are due to the reverse fragment adding order and the fact that the last allocated page will have a hole R with an average size of half a page in it. Such a good spatial locality resulting from gnu malloc justifies our contiguous assumption. It is also possible to enhance the malloc routine to allocate PDS nodes that are circularly contiguous in the cache; one might use this to simulate contiguous allocations in cases when intervening allocations of other similar-sized PDS nodes are unavoidable.

3 Cache Miss Analysis and Validation

In this section, we elaborate on our cache miss analysis for linked lists and binary trees, starting first with observations about the programs, and then giving our analysis and simulation validations.

3.1 Program Patterns and Assumptions

The usual reference pattern of linked list and binary tree programs is composed of traversals, either complete or partial. Traversals can be further categorized as build

(allocate) or compute. The build traversal allocates memory for each node and links them together. The compute traversal processes information contained in each node, i.e. doing some computation for each of the traversed node.

For linked lists, we assume that the allocate traversal is done backwards: that is, new nodes are added in the front of the list. To improve cache performance, the compute traversal is done forwards, so successive traversals alternate their orders. See Figure 1 for a C code snippet for the linked list example.

```
typedef struct list {
  long int val;
  struct list *next;
} list_t;

list_t *build()
{
  int i=0;
  list_t *h,*p;
  h=NULL;
  while(i<NUM) {
       p=(list_t *) malloc(sizeof(list_t));
       p->val=i;
       p->next=h;
       h=p; i++; }
  return (h);
}

compute(list_t *p)
{
  while(p) {
       p->val+=NUM;
       p=p->next; }
}
```

Code for linked list

```
typedef struct tree {
  long int val;
  struct tree *left;
  struct tree *right;
} tree_t;

tree_t *build(int i)
{
  tree_t *new,*left,*right;
  if(i) {
     new=(tree_t *) malloc(sizeof(tree_t));
     left=build(i-1);
     right=build(i-1);
     new->val=i;
     new->left=left;
     new->right=right;
     return new; }
  else return NULL;
}

void compute(tree_t *r)
{
  if(r) {
     r->val++;
     compute(r->left);
     compute(r->right); }
}
```

Code for binary tree

Figure 1: C Code Snippet

For binary tree, the build and compute phases are both pre-order depth-first traversals. Pre-order depth-first traversal simply means that the current node is processed first, then the left subtree and finally the right subtree. See Figure 2 for a diagram and Figure 1 for a C code snippet for the binary tree.

We first assume that all nodes are allocated contiguously in the heap memory. The previous discussion of the gnu malloc implementation justifies this assumption, as this is the natural result of either a complete or a segmented build traversal with no intervening mallocs from the same freelist. We also make the simplifying assumption that every traversal accesses all elements contained within each PDS node.

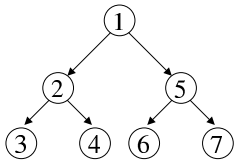

Figure 2: Diagram of Binary Tree

3.2 Cache Miss Analysis

We use notation as follows. We consider direct-mapped caches of size C_S. B_S represents cache block size, which is assumed to be always in the powers of two; N_n and N_S designate number of nodes and node size, respectively. Finally, Data Set Size is abbreviated as DSS, and $DSS = N_N * 2^{\lceil log_2^{N_S} \rceil}$. For binary trees, $N_N = 2^L - 1$, where L is the number of tree levels.

3.2.1 Cold Misses

The entire data set will suffer cold misses during the build traversal.

- $N_S \leq B_S$: $ColdMisses = \frac{DSS}{B_S} + \frac{HeapinfoSize}{B_S}$
- $N_S > B_S$: $ColdMisses = N_N * \lceil \frac{N_S}{B_S} \rceil + \frac{HeapinfoSize}{B_S}$

The second part of both expressions is due to initializing the malloc bookkeeping data structures called heapinfo at the first call of malloc. The heapinfo size is initially 16K for gnu malloc and this resides in the heap itself.

3.2.2 Replacement Misses

All the cold misses will be incurred during the build traversal. Following this, the compute traversal will either experience cache hits, on items that still remain in cache, or replacement misses, on items that have been evicted. Items that remain in cache after the build traversal are referred to here as the cache residue. For other nodes not in the cache residue, a small number of them are evicted by stack references (whose effect are neglected here) but most of them are evicted by accesses to other list items in the heap.

1. $DSS \leq C_S$

   If the data set size is less than the cache size, the cache residue is always fully consumed, and clearly there will be no replacement misses during the compute traversal. This is true whether the traversal order alternates or not.

2. $C_S < DSS \leq 2C_S$
   See Figure 3 for illustration, keeping our contiguity assumption in mind.

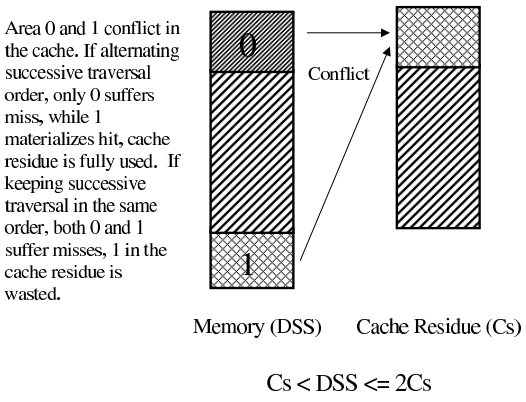

Figure 3: Cache Mapping when $C_S < DSS \leq 2C_S$

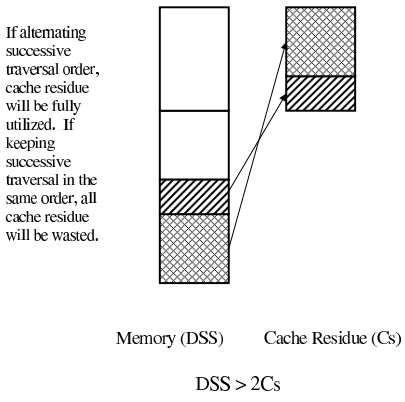

Figure 4: Cache Mapping when $DSS > 2C_S$

For binary tree, the compute traversal order keeps the same as build traversal, the cache residue is only partially consumed, both area 0 and area 1 (and both of their size equal $DSS - C_S$) will suffer cache misses.

There are two subcases.

- $N_S \leq B_S$: $ReplMisses = \frac{DSS - C_S}{B_S} * 2$
  When $N_S \leq B_S$ the cache is big enough to hold the biggest left subtree. This allows spatial locality to be exploited when accessing the right sub-tree after finishing processing its value and traversing its left sub-tree.
- $N_S > B_S$: $ReplMisses = (N_N - \frac{C_S}{2^{\lceil log_2^{N_S} \rceil}}) * \lceil \frac{N_S}{B_S} \rceil * 2$

For a linked list, the compute alternates traversal order compared with build, so the cache residue is guaranteed to be fully consumed. Therefore only area 0 from Figure 3 will suffer misses. This reduces the cache misses by half for both of the two subcases above.

3. $DSS > 2C_S$

Consider first a binary tree.

- $N_S \leq B_S$

$$ReplMisses = \frac{DSS}{B_S} + \frac{(2^{(L - \lfloor log_2^{(\frac{C_S}{B_S}+1)} \rfloor)} - 1) * 2^{\lceil log_2^{N_S} \rceil}}{B_S}$$
$$= \frac{2^{\lceil log_2^{N_S} \rceil}}{B_S} * ((2^L - 1) + (2^{(L - \lfloor log_2^{(\frac{C_S}{B_S}+1)} \rfloor)} - 1)) \quad (1)$$

The second item is added by those tree nodes in the top subtree whose left sub-trees are beyond the cache size. These nodes incur misses when traversing their right sub-trees after finishing their left sub-tree's traversal, causing the potential spatial locality to be unrealized.

- $N_S > B_S$ $ReplMisses = N_N * \lceil \frac{N_S}{B_S} \rceil$

For linked list, because of the traversal order alternation, the cache residue is fully consumed, so the miss number is reduced in correspondence to the cache size as shown below.

- $N_S \leq B_S$: $ReplMisses = \frac{DSS - C_S}{B_S}$
- $N_S > B_S$: $ReplMisses = (N_N - \frac{C_S}{2^{\lceil log_2 N_S \rceil}}) * \lceil \frac{N_S}{B_S} \rceil$

3.3 Simulation Validation

3.3.1 Simulation Methodology

We simulate the C code for both linked list and binary tree on a Compaq Alpha-based machine using Compaq's ATOM tool [16]. Through ATOM, we build an instrumented version of our program, which simulates a single level data cache for every data load and store, and gathers statistics of both total cache misses and heap cache misses.

3.3.2 Simulation Results

The statistics shows that heap misses constitute over 96% of the total misses for linked list, and over 93% of the total misses for binary tree. Here, we compare heap miss numbers with our mathematical analysis. We simulate three different combinations of cache configurations and program parameters for linked lists (Figures 5, 6 and 7) and two for binary trees (Figures 8 and 9).

The x-axis on the graphs of binary tree is the number of tree levels rather than tree nodes, which gives a logarithmic effect on our curves.

The graphs show that our mathematical analysis (line) matches the simulation results (points) well. There is a little under-estimation for our mathematical analysis, which is understandable since we do not take the effect of all auxiliary data (e.g. global data) into consideration.

4 Piecewise Linear Layout

Until now, this paper has made the simplifying assumption that PDS nodes are allocated contiguously. At this point, we relax our contiguity assumption by considering piecewise linear layouts with occasional skips between items. There are two main cases:

- Intra-page contiguous with only big inter-page skips in step of $s*pagesize (s \geq 0)$

  This results from allocation which has other intervening allocation of different malloc sizes lying in between. The other intervening allocations will causes pages

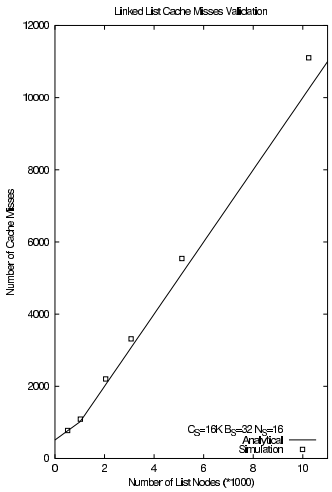

Figure 5: Validation of Linked List: $C_S = 16K, B_S = 32, N_S = 16$

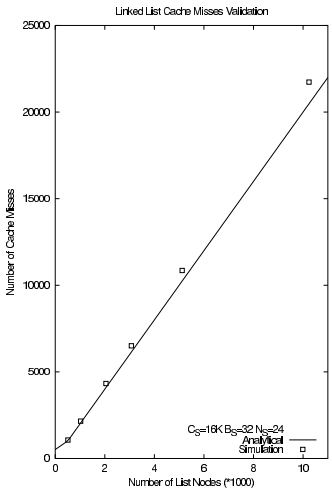

Figure 6: Validation of Linked List: $C_S = 16K, B_S = 32, N_S = 24$

to be devoted to freelists for other malloc sizes. A similar skip patter will also occur in cases where the pages might be contiguous in virtual memory, but are not contiguous in physical memory.

- Small skips in step of $s * 2^{\lceil log_2^{N_S} \rceil}$ across the whole layout

  This results from allocations which have intervening allocations off the same freelist.

Clearly, these two cases can also appear within the same allocation stream, but we consider them separately here.

4.1 Cache Miss Analysis for Piecewise Contiguity

Big Skips

Big skips will impact cache temporal locality, leaving spatial locality unchanged. These skips impact temporal locality mainly by spoiling cache residue consumption. This will occur when the skip pattern leads to hole in the cache residue, but its probability is quite low if $DSS \gg C_S$. For case where $DSS \gg C_S$ does not hold, it is not difficult to get the cache residue layout given the big skip patterns.

Small Skips

Small skips mainly spoil spatial locality, as opposed to temporal locality. They mainly affect the case when $N_S \leq B_S$. If $N_S \leq B_S$ and $\frac{B_S}{2^{\lceil log_2^{N_S} \rceil}} = 2^m (m > 0)$, suppose the

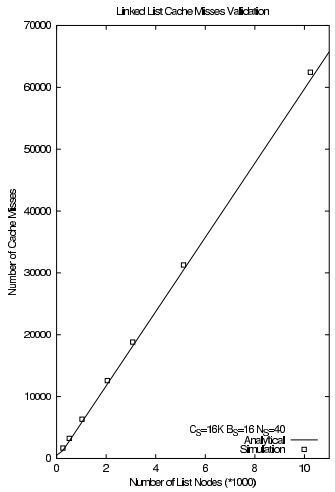

Figure 7: Validation of Linked List: $C_S = 16K, B_S = 16, N_S = 40$

skip step is $s * 2^{\lceil log_2^{N_S} \rceil}$. If $s + 1 < 2^m$, then

$$ColdMisses = \frac{(s+1)*N_N*2^{\lceil log_2^{N_S} \rceil}}{B_S} + \frac{HeapinfoSize}{B_S}$$

$$ReplMisses = \frac{(s+1)*N_N*2^{\lceil log_2^{N_S} \rceil} - C_S}{B_S} \quad (2)$$

(3)

On the other hand, if $s + 1 \geq 2^m$, then

$$ColdMisses = N_N + \frac{HeapinfoSize}{B_S} \quad (4)$$

$$ReplMisses = N_N - \frac{C_S}{B_S} * \frac{2^m}{s+1} \quad (5)$$

For $N_S > B_S$, small skips only affect the cache residue by adding small holes in it; this effect is small enough to be neglected.

5 Related Work

This section now relates the analysis work we have begun to prior work for both regular and irregular programs. Ghosh et. al. proposed a CME mathematical framework for array loop nest [6]. While effective, this framework is mainly applicable to array-based codes with regular access patterns.

In recent years, cache behavior research for pointer data structures has increased. Chilimbi et. al., proposed a cache conscious malloc routine, cache conscious garbage collector, and run-time memory reorganizer [2, 3, 4]. Grunwald et. al. also studied the cache locality issue of various malloc implementations [7]. Ding et. al. studied

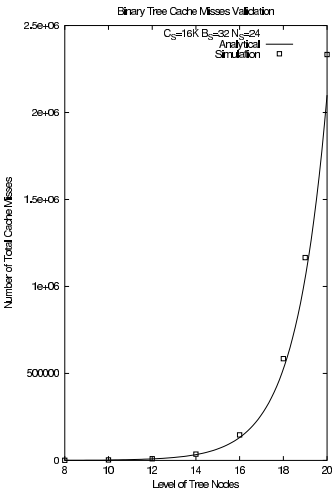

Figure 8: Validation of Binary Tree: $C_S = 16K, B_S = 32, N_S = 24$

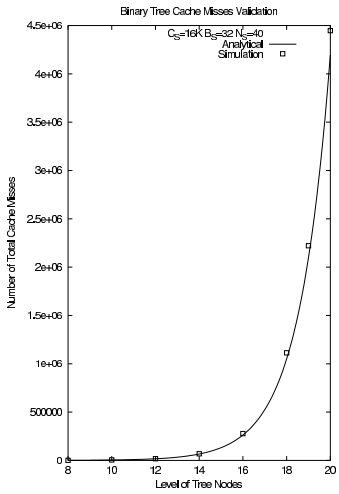

Figure 9: Validation of Binary Tree: $C_S = 16K, B_S = 32, N_S = 40$

both run-time data and computation reorganization for PDS applications [5]. Calder et. al. proposed a dynamic cache conscious data placement optimizing algorithm using profiling, which takes all the concerning segments including heap, stack and global data section into consideration [1]. Both Chilimbi's cache conscious garbage collector and Calder's cache conscious data placement optimizer use software profiling and a certain graph to record the profiling information, which is then used to guide the later data placement. Kistler and Franz applied the similar ideas to gather the spatial locality information by the so-called temporal relationship graph to guide the intra cache line placement on a much finer granularity by exploiting memory interleaving and cache line-fill buffer forwarding characteristics [9]. Kumar and Wilkerson apply spatial locality prediction hardware to pick up cache line components dynamically based on the profiling history [10]. All the above techniques for PDS either apply heuristics or use profiling or both to accomplish the cache conscious effect.

Our ongoing efforts focus on using the elementary analysis described here as a foundation to help us analyze the cache behavior of database applications. Database applications uses pointer-based structures intensively, and some work has been done on analyzing and optimizing their performance. Trancoso et. al. characterized cache behavior of database application under DSS Commercial Workloads in a multi-processor environment [17]. They also proposed a cache conscious query plan generator for database application and applied some traditional cache miss reduction method, i.e. , data blocking, software prefetching in the database setting [18]. Rao and Ross proposed Cache Sensitive Search Tree to substitute for B^+ Tree as key indexing structures, but only limited to DSS workload where data input and modification are rare and data modification can be done in batch [13, 14]. Karlsson et. al. devised an analytical database cache miss model for a DSS workload, using the working set concept [8]. Our

work seeks to provide a more detailed extension of their initial approach.

6 Conclusions

By carefully analyzing the memory reference pattern of PDS centric programs, we have derive some analytical representations of cache miss numbers for linked lists and binary trees. We plan to use these analytic expressions as a foundation for understanding programs whose key operations manipulate trees and lists. A general optimization strategy is to order computation traversals such that they always seek to consume the cache residue before bringing in new items to the cache. We plan to build on this work in two main ways. First, we plan to extend our models to relax assumptions about memory allocator behavior. In particular, we will use probabilistic analysis to extend on our piecewise linear skip assumptions from Section 4. Second, we plan to compose these expressions together into more complicated descriptions of database and other complex applications.

References

[1] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1998.

[2] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, May 1999.

[3] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, May 1999.

[4] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In International Symposium on Memory Management, Oct. 1998.

[5] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, May 1999.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems, July 1999.

[7] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, June 1993.

[8] M. Karlsson, F. Dahlgren, and P. Stenstrom. An analytical model of the working-set sizes in decision-support systems. In International Conference on Measurement and Modeling of Computer Systems, June 2000.

[9] T. Kistler and M. Franz. Automated record layout for dynamic data structures. Tech. Report 98-22, University of California at Irvine,Department of Information and Computer Science, May 1998.

[10] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In the 25th Annual International Symposium on Computer Architecture, June 1998.

[11] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1996.

[12] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems, July 1996.

[13] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In International Conference on Very Large Data Bases, Sept. 1999.

[14] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In International Conference on Management of Data and Symposium on Principles of Database Systems, May 2000.

[15] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1998.

[16] A. Srivastava and A. Eustace. ATOM A System for Building Customized Program Analysis Tools. In Proc. PLDI, June 1994.

[17] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of dss commercial workloads in shared-memory multiprocessors. In Third International Symposium on High-Performance Computer Architecture, Jan. 1997.

[18] P. Trancoso and J. Torrellas. Cache optimization for memory-resident decision support commercial workloads. In International Conference on Computer Design, Oct. 1999.