Tracer: Monitoring Fine Grained Memory Access

Ravi Tandon, Jonathan Balkind, Nevin Li

Abstract

Memory intensive applications such as *memcache, web server accelerators, etc.* require large amounts of memory. However, these applications have a highly skewed memory access footprint in such that a small subset of objects comprise most of the memory accesses. In this paper, we present the design and implementation of Tracer: a tool that allows developers to trace memory accesses at object level granularity. The design philosophy of Tracer is to assist such applications in identifying *entities* (such as C level *structures*) which get heavily accessed. Tracer accomplishes this with negligible intrusion into the application code, allowing for easy integration into existing projects. We evaluated a prototype implementation of Tracer and observed reasonable overheads of 28% and 40% over non-modified C code for creation and read heavy workloads. As compared to a page protection mechanism, Tracer improves the average performance by 12% and 6.28% for read and update heavy workloads.

1 Introduction

Memory tracing is a difficult problem to resolve since most of the accesses to memory occur through hardware (in traditional C based applications). Software based techniques such as those used by *Valgrind's memcheck [5]* and *gdb's ptrace* [1] rely on memory protection mechanisms which introduce heavy overheads and therefore cannot be effectively used with high end applications. This work presents the design, implementation and evaluation of Tracer, a code based instrumentation system coupled with memory monitoring techniques which enables applications to transparently obtain application's memory footprints with low overhead.

Design philosophy Tracer provides a APIs for efficient tracking of object-level memory accesses. Previous works on object caching makes use of memory protection mechanisms, resulting in page faults on object accesses and high virtual memory space utilization. We aim to minimize overheads while providing useful information to the application programmer. This information can assist application developers in enhancing application performance by compact placement of "hot" objects. This would reduce access latencies and result in better memory usage. Overall, the philosophy underlying the design of Tracer's architecture can be enumerated as follows:

1. The interception mechanism must be completely transparent to the application developer.

2. The interceptor and the memory monitoring libraries must be easily pluggable into a standard C application.

3. The overall overheads must be reasonable compared to the performance of a vanilla C application.

4. The design of the solution must be system independent (independent of POSIX system calls such as protection mechanisms, underlying hardware).

This work presents the design of Tracer. Tracer provides a C pre-processing tool called the *interceptor* and a set of APIs (bundled with the standard GNU C library) for monitoring and providing the overall memory access footprint at an object level granularity to C application developers. Our system design achieves all the above lying objectives (outlined *design philosophy*. The interceptor uses preprocessing through standard code parsing techniques and injects *"memory access"* calls into the C program, thus making the design transparent to the application developer and easily pluggable with standard C applications. The monitoring library is implemented as a part of the standard GNU C library. The solution does not require additional hardware and is therefore independent of the platform used. Our implementation on binary tree micro benchmarks report an overhead between roughly 20% and 40% for create and read workloads. We, therefore, believe that Tracer is a useful and efficient software solution which could augment application development through better memory monitoring techniques.

The paper is organized as follows: Section 3 describes the design of Tracer. Section 4 describes the microbenchmark study and discussion of results. Section 2 outlines the motivation underlying our work. Section 6 discusses some of the works related to Tracer's design. Section 5 provides insight into useful ideas which could improve the design of Tracer. Section 7 provides a summary of our work.

2 Motivation

Memory Augmentation Modern web-scale applications (web accelerators, proxy servers, in-memory key-value stores) tend to rely heavily on data (indices, key-value pairs) cached in DRAM as their primary storage for faster access. Improving the utilization of the available DRAM can reduce the dependency on HDDs, SSDs. Additionally, alternate memory solutions such as Storage Class Memories (SCM) could augment the DRAM either by being available as disk caches or slower DRAMs.

Memory Tiering Efficient placement of data at different levels within the memory hierarchy is referred to as *memory tiering*. One of the key requirements of *memory tiering* would be *transparent* movement of application data between the different layers of hierarchy. The adaptability of such *hybrid memory hierarchies* would depend on the unobtrusiveness of software technologies which support these memories. Tiering data between memory hierarchies would require an understanding of memory access patterns.

Memory Tracing *Memory tracing* techniques monitor application data at finer granularity and provide insights into better memory management techniques. *Memory tracing* techniques such as *memcheck, ptrace, data flow tracking mechanisms (DFTs), binary rewriting mechanisms [6, 7]* introduce higher overheads. This work explores code instrumentation and object tagging mechanisms in order to build efficient memory monitoring libraries.

3 Design

High level design Tracer is a designed as a generic pluggable tool for monitoring object level accesses for C programs. The high level design of Tracer can be bifurcated into two major components. These consist of an *interception* based pre processing engine and a *monitoring engine* coupled with the standard memory management library (*malloc library*. Besides, Tracer provides flexible interfaces to the developer to monitor object access levels accurately.

Flow of operations The flow of operations which eventually leads to memory monitoring can be summarized in the following three steps (refer to Fig. 1):

1. A preprocessing engine which instruments C code with memory access calls.

2. A set of standard APIs which provide interfaces which:
   - allocate/deallocate objects augmented with headers,
   - register memory access calls
   - output summary of memory access footprint at object level granularity

3. A statistics store for saving application's object access count information.

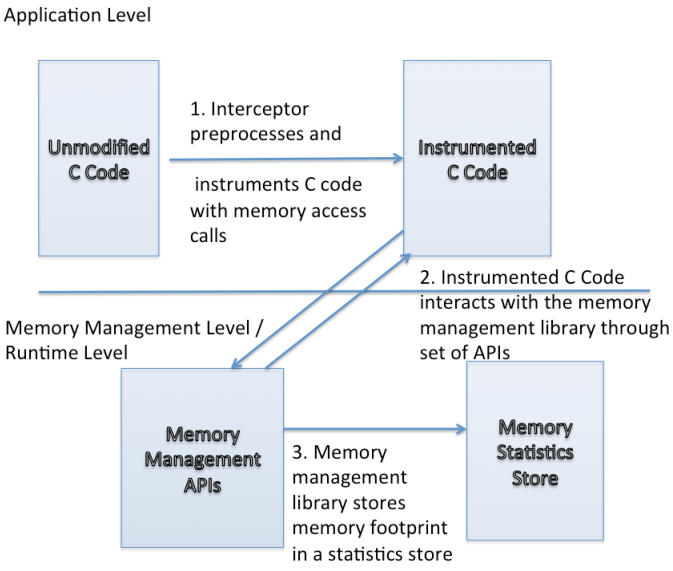

Figure 1: High level architectural design

Interceptor: Transparent Instrumentation The interceptor uses preprocessing and code level instrumentation for injecting calls which enable the monitoring mechanism. In order to monitor object references, Tracer uses an interceptor which pre-processes standard C source code. The interceptor classifies literals into different classes of 8 different class types. *Identifiers* which are allocated as dynamic objects are identified as *trace targets*. The interceptor adds an *access* function call (*mem_access*) following an access to each *trace target*. This *mem_access* library call increments the object's hidden count variable by one.

Header Augmented Objects Tracer implements a custom version of the standard malloc library function named "*hmalloc*" (header memory allocator) which *transparently* inserts the target object with header field. The header acts as custom metadata store for the object. The design philosophy behind augmenting objects with headers is to be able to support faster updates. Our earlier design was plagued by the heavy overheads involved in looking up each objects counter within a hash table. *Object headers* provide a clean solution to this problem since the metadata is now tagged to each object in memory. *Hmalloc*

function increases the amount of bytes requested by the size of an int, and uses the additional 4 bytes allocated to store the object count. A pointer to the memory location immediately following these four bytes, is then returned to the requesting program. These objects are then added to an object list that Tracer manages. It is a part of memory statistics interfaces (described in the next paragraph).

Memory Statistics Interfaces We provide the following interfaces to the application developer for getting memory statistics:

1. *void * hmalloc (size_t bytes)* - This interface allocates "bytes" size object tagged with a header
2. *void mem_access_stat* - This interface prints the access count of all the objects allocated by the current program
3. *void hfree* - frees objects tagged with headers
4. *void mem_access(void *memory_location, int count)* - This interface increases the access count of the object at the memory location (memory_location) by the count. This is the interface which is used by the memory interceptor to monitor object level access count
5. *int mem_access_count(void *object_address)* - This interface returns the current access count of the object (at memory location object_address).

4 Evaluation

In order to measure the performance of the we perform a microbenchmark study on the B-trees. We perform four sets of tests for our microbenchmark study. We measure the overall time taken for the creation of nodes of a large binary tree. We also test the overheads involved in the search, update and traversal operations. Our experiments were done on a Intel *i5* virtual machine, supporting *2.5 GHz* processor speed. The DRAM size allocated to the virtual machine is *4GB*. All the experimental results have been averaged over 10 different runs.

We perform a comparative analysis and compare Tracers overhead with the performance of the implementation of a binary tree using an unmodified implementation in C (we call it *vanilla C implementation*). We also implement a page protection mechanism, similar to the mechanism used in [2] for tracking objects. Our primary objective behind the study is to measure the overall overhead of our memory monitoring mechanism and compare it with a page protection mechanism. The implementation of the page protection mechanism uses a page buffer size of *25MB* (as suggested in [2]).

B-Tree Creation Tests The first test measures the overall overhead that Tracer has while monitoring the access to objects. We create trees containing nodes ranging from 2500 to 25000 nodes. The experimental results indicate that Tracer invokes an overhead of around 20% over vanilla C implementation. Page protection mechanism has an overhead of 3 orders of magnitude (refer to Fig. 2). The average time for creation of a binary tree with vanilla C implementation is around 3.77 milliseconds (over the different number of nodes), 4.5 milliseconds for Tracer and over 1 second for page protection mechanism. One of the primary reasons for the poor performance of page protection mechanism is the overheads involved in the eviction of an object from the page table and its subsequent insertion in the object table. This requires a system call (triggered due to access to a protected page), eviction of page from the page buffer (overheads due to memory copy), page materialization (requiring memory copy from the object table to the page buffer and a look up in the object table).

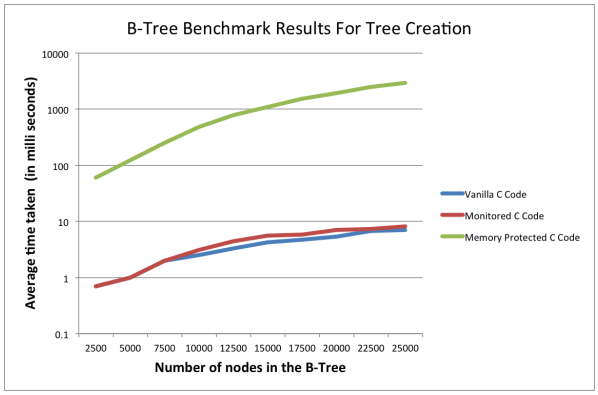

Figure 2: B-Tree Benchmark Results For Tree Creation

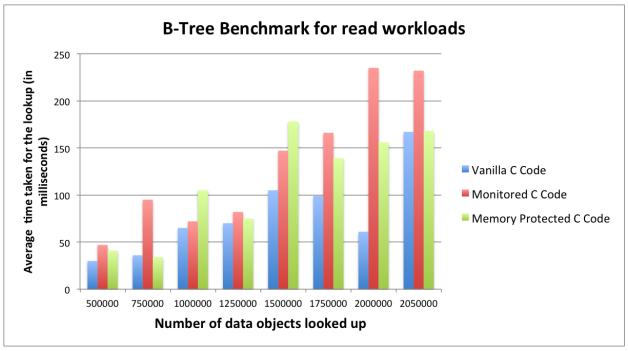

Figure 3: B-Tree Benchmark Results For Read Workloads

B-Tree Read Tests The second set of tests measures the overall overhead incurred in reading a set of random ob-

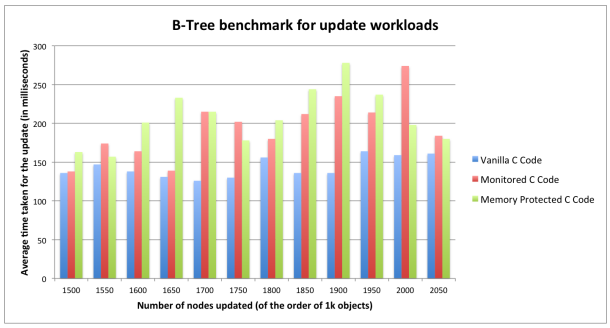

Figure 4: B-Tree Benchmark Results For Update Workloads

jects from a B-Tree of a fixed size (number of nodes in the binary tree was kept to 2500). For the read tests, we generate a random number of keys and search them in a preconstructed binary tree. We vary the number of keys read from 500, 000 to 2 million per experiment. Each data point is an average of 10 different experiments. Tracer has a reasonable overhead (of around 40%) over the vanilla C implementation. Page protection mechanism has an overhead of 28% (refer to Fig. 3). Tracer performs a fine grained monitoring, whereas page protection mechanisms provide an approximation of the access pattern of the application. Subsequently, Tracer has slightly larger overheads over page protected mechanisms. However, these overheads are reasonable considering the information which our tracing tool provides the developer with.

B-Tree Update Tests The third set of tests measure the overall overhead incurred for update intensive workloads. As in the read test, we generate a random set of key pairs. Each key pair has a source and a destination key. The source key is searched in the binary tree and its value is updated with the destination keys value. If the source key is not found in the binary tree the update procedure returns. We vary the number of update operations from 1.5 million to 2.5 million for our tests. Tracer improves the performance of the update operations over page protected mechanism by 6.28%. The overhead over a vanilla C implementation is around 35% These tests indicate that Tracer performs well in comparison to page protection mechanism for update heavy workloads.

B-Tree Traversal Tests The fourth set of experiments explore the performance of tree traversals for the three different mechanisms. Tree traversals are generally observed heavily for range queries in databases. Page protected mechanisms perform poorly for such workloads with over 2 orders of magnitude of overheads.

The observation from the tree traversal and creation tests is that page protection mechanisms perform poorly when a large number of nodes of the tree are touched. This could be explained because such an access pattern would result in a high "*object flux*" between the page buffer and page table. The movement of an object between the page table and page buffer involves page eviction and page materialization which have high overheads. Tracer circumvents these overheads by its interception and tagged counter based approach.

5 Future Work

We envision Tracer to be capable of supporting efficient memory management libraries which would be wrapped around it. Tracer provides accurate memory access pattern to the application developer. This information could be used in offline and online memory management models.

5.1 Online Memory Management Model

The online memory model design would involve a dynamic memory manager (similar to a Java Run Time) which would move objects and pack objects with similar access patterns together in same pages. The dynamic memory management system built would be built on top of Tracer. The caching system used in SSDAlloc is able to change the location of objects owing to its *"Object Per Page Model"*. However, such a design increases TLB pressure. In keeping with our model of simplicity in implementation and a focus on user space solutions, we would prefer to take a different path. However, moving objects in the virtual memory requires updating any pointers to those objects, and as such would require backlinks unless application semantics were significantly restricted. As such, we expect that the dynamic memory management implementation would make user of programmer's cooperation, in a manner akin to cooperative multithreading. Here, applications would indicate when they were in a state that would guarantee that object pointers are currently not being accessed. This would activate our memory management system, freezing the application until suitable objects were relocated.

5.2 Offline Memory Management Model

In the offline memory management model, the application developer can run benchmarks on application design. Tracer would provide fine grained access information. The developer could then enhance the application's performance by adequately restructuring application's memory usage (by allocating frequently accessed objects together).

Improvement in interceptor's capabilities The first of these would be increasing the capability of the intercep-

tor's parsing abilities. At the moment, the subset of C which the parser supports focuses on structs and simple C syntax, but in order to support more idiomatic C, more features would have to be added. Primary amongst these is support for arrays. Since, individual structural units within an array represent related semantic information, it may be useful to have them packed together as a single unit. Such a design model would be compatible with pointer arithmetics within arrays too.

6 Related Work

Fine grained access mechanism [3] and SSDAlloc [2] use an *OPP(object per page)* model and *page protection* mechanism to monitor fine grained access patterns. *HAC (Hybrid Adaptive Caching for Distributed Storage Systems)* [4] is a dynamic caching system which uses *indirection* mechanism to refer to smaller 4 byte objects. Indirection technique (those using handles) trade off transparency for simplicity (from the system designer's perspective).

Description of Chameleon and SSDAlloc *SSDAlloc*

SSDAlloc monitors fine grained access patterns through the use of the Object Per Page model, where each object is placed in its own page of virtual memory. SSDAlloc's DRAM is split into an object cache (which composes most of DRAM), and a Page Buffer that holds the set of materialized pages currently being used by the application (where each object is stored in its own physical page). SSDAlloc protects all virtual memory that it allocates, so any memory access for an unmaterialized page generates a page fault and is sent to SSDAlloc's interrupt handler. The handler would then pull the object from the object cache or the SSD, unprotect it and materialize it in the page buffer, and send it back to the application. This allows SSDAlloc to track memory accesses at an object level granularity as opposed to a page level granularity, since each virtual memory page access can be directly mapped to a single object access. The downside to this approach is that since materialized pages in the page buffer are unprotected, SSDAlloc cannot track memory accesses for materialized pages. This restriction resulted in SSDAlloc implementing a FIFO eviction policy for the page buffer, since LRU is infeasible if materialized page accesses cannot be monitored. SSDAlloc can be modified to page fault for materialized pages, but it isn't well suited for it since doing so would significantly affect performance due to the increased amount of page fault handling. Tracer allows the user to track the total number of object accesses, which may prove useful when implementing policies such as a LRU scheduler.

Chameleon [3] improves on SSDAlloc's design by removing the need to split DRAM into a cache and page buffer entirely. Each object is still put into a single virtual page, but the virtual pages are partitioned into a set of object sized chunks, and the object is randomly placed into one of these chunks in the virtual page. Since these chunks can be mapped to a chunks in a physical page, a single physical page can store objects from multiple virtual pages. These virtual pages are tracked in two FIFO queues, one for active pages and one for inactive pages. In order to track object usage, Chameleon uses a kernel module that modifies the page table, and more specifically updates a "recency bit" whenever an object is accessed. Whenever a new page is added to the active queue, Chameleon checks that bit for the tail of the queue, and if it hasn't been set, moves it to the inactive queue. If the system needs to evict a page, it checks the tail of the inactive queue, and evicts the first page that hasn't had that bit reset. One issue here is that Chameleon isn't as transparent as Tracer in that it needs to install a kernel module in order to function properly.

7 Conclusion

This paper presents Tracer, an instrumentation based lightweight, transparent memory tracing mechanism for C applications. This technique instruments object accesses with additional library calls, augments object structures with transparent headers and provides useful interfaces to application developers for getting useful information about memory access patterns. We also implement a memory protection based mechanism based on SSDAlloc [2] and perform microbenchmark studies binary trees. We evaluated a prototype implementation of Tracer and observed reasonable overheads of 28% and 40% over non-modified C code for creation and read heavy workloads. As compared to the page protection mechanism, Tracer improves the average performance by 12% and 6.28% for read and update heavy workloads. We observe that for create and traversal workloads, tracer results in a reduction of 2-3 orders of magnitude over page protection mechanisms. This is because creation and traversal workloads have memory footprints larger than available page buffer sizes and page protection mechanisms (such as SSDAlloc), incur heavy overheads in control transfer to kernel due to page protection, page eviction and materialization.

The source code of Tracer's prototype is available at `http://bit.ly/1gvF1Mk`.

References

[1] URL "`http://aosabook.org/en/gdb.html`".

[2] A. Badam and V. S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX*

conference on Networked systems design and implementation, pages 16–16. USENIX Association, 2011.

[3] A. Badam and V. S. Pai. Better flash access via shape-shifting virtual memory pages. In *Proc. Conference on Timely Results in Operating Systems*, Farmington, PA, Nov. 2013.

[4] M. Castro, A. Adya, B. Liskov, and A. C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 102–115. ACM, 1997.

[5] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[6] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[7] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing.