Tracer: Monitoring Fine Grained Memory Access

Ravi Tandon, Jonathan Balkind, Nevin Li

Abstract

High performance applications such as *in-memory key-value stores, web server accelerators, etc.* require large amounts of memory. However, these applications have heavy tailed memory access footprints. 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 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 with existing projects. We evaluated a prototype implementation of Tracer and observed only reasonable overheads of 28% and 40% over non-modified C code for creation and read heavy workloads, respectively. As compared to a page protection mechanism, Tracer improves the average performance by 12% and 6.28% for read and update heavy workloads, repectively.

1 Introduction

For traditional C based high-end applications, memory tracing is a difficult problem to resolve since most of the accesses to memory occur through hardware. Software based techniques such as those used by *Valgrind's memcheck [8]* and *gdb's ptrace* [1] rely on memory protection mechanisms which introduce heavy overheads. Therefore, these techniques cannot be effectively used for monitoring high performance 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 APIs for efficient tracking of object-level memory accesses. Previous works on object caching make 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 is outlined 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 (an uninstrumented C application).
4. The design must be system independent (i.e. independent of POSIX system calls such as protection mechanisms and reliance on system dependent hardware or architecture).

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. Tracer achieves all the above lying objectives (outlined in *design philosophy*. The interceptor uses preprocessing using standard code parsing techniques and injects *"memlets (memory access calls)"* into the C program, thus making the design transparent to the application developer and easily pluggable with a standard C applications. The monitoring library is implemented as a part of the standard GNU C library. The solution does not depend on hardware or 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, repectively. We, therefore, believe that Tracer is a useful and efficient software solution which could assist application development by providing better memory monitoring.

The paper is organized as follows: Section 2 outlines the motivation underlying our work. Section 3 describes the design of Tracer. Section 4 describes the microbenchmark study and discussion of results. Section 5 provides insight into useful ideas which could improve the design of Tracer. Section 6 discusses some of the previous pieces

of work related to Tracer's design. 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) rely heavily on data (indices, key-value pairs) cached in DRAM as their primary storage. Improvement in the utilization of available DRAM can reduce the latencies incurred from accessing seconadry stores (SSDs, HDDs). Determining which objects are most frequently accessed would allow developers to pack those objects in DRAM and minimize latency due to disk accesses. Additionally, alternative memory solutions such as *Storage Class Memories* (such as memristors, phase change memory, etc.) [7] would augment the DRAM by being available either 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* is *transparency* in the movement of application's data between different layers of memory hierarchy. The adaptability of such *hybrid memory systems* would depend on the unobtrusiveness of software technologies supporting them. Tiering data in memory hierarchies requires an understanding of memory access patterns that is unlikely to be clear to the programmer from the application's design alone.

Memory Tracing *Memory tracing* techniques monitor application data at a fine granularity and provide insights into better memory management. Existing*Memory tracing* techniques such as *memcheck, ptrace, data flow tracking mechanisms (DFTs), binary translation mechanisms [9, 10]* introduce high overheads, and so finding new techniques with smaller overheads can help improve an application's performance.

3 Design

High level design Tracer is designed as a generic pluggable tool for monitoring object level accesses for C programs. The high level design of Tracer can be split into two major components. Tracer consists of an *interception* based preprocessing engine and a *memory monitoring engine* (coupled with a statistics store). Tracer provides the developer with flexible interfaces in order to monitor object access levels accurately.

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

1. A preprocessing engine instruments C code with memlets (memory access calls).
2. A set of standard APIs provides interfaces which:
   - allocate/deallocate objects augmented with headers
   - register memory access calls
   - provide a summary of memory access footprint at object-level granularity
3. A statistics store saves the application's object access count information.

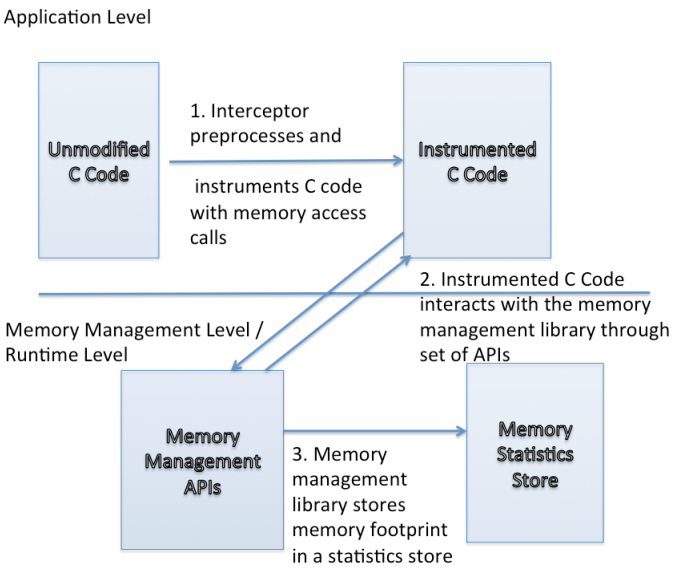

Figure 1: High level architectural design

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* 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 Tagged Objects Tracer implements a custom version of the standard malloc library function named "*hmalloc*" (header memory allocator) which *transparently* tags each target object with a header field. The header acts as a custom metadata store for the object. The design philosophy behind tagging objects with headers is to be able to support faster count updates. Our earlier design

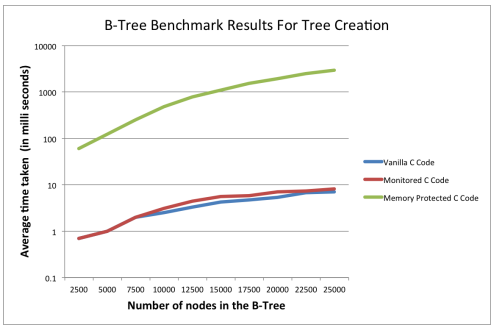

Figure 2: Binary Tree Benchmark Results For Tree Creation

was plagued by 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* increases the amount of bytes requested by the size of an int, and uses the additional space allocated to store the object count. A pointer to the memory location immediately following these four bytes is then returned to the requesting application. These objects are then added to an object list that Tracer manages, which is a part of the memory statistics store (described below).

Memory Monitoring Interfaces We provide the following interface to the application developer for getting memory statistics:

1. *void * hmalloc (size_t bytes)* - Allocates "bytes" size object tagged with a header.
2. *void hfree* - Frees objects tagged with headers.
3. *void mem_access_stat* - Prints the access count of all the objects allocated by the current program.
4. *void mem_access(void *memory_location, int count)* - Increases the access count of the object at the memory location (memory_location) by the count. This function is used by the memory interceptor to monitor access counts at the object-level.
5. *int mem_access_count(void *object_address)* - Returns the current access count of the object (at memory location "object_address").

4 Evaluation

In order to measure the performance of Tracer, we perform a microbenchmark study using binary trees as an exemplar in-memory data structure. 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 for search, update and traversal operations. Our experiments have been done on an Intel *i*5 virtual machine with a *2.5 GHz* clock frequency. The DRAM size allocated to the virtual machine is *4GB*. All the experimental results have been averaged over 10 different runs.

In our analysis, we compare Tracer's overhead with the performance of an unmodified implementation in C (we call it the *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].

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

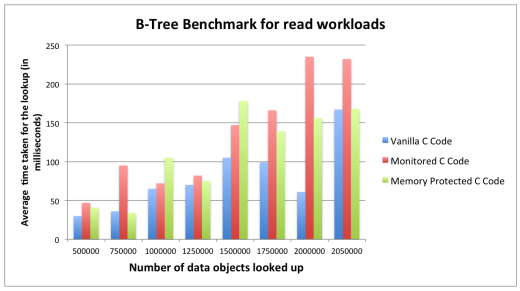

Figure 3: Binary Tree Benchmark Results For Read Workloads

Binary Tree Read Tests The second set of tests measure the overall overhead incurred in reading a set of ran-

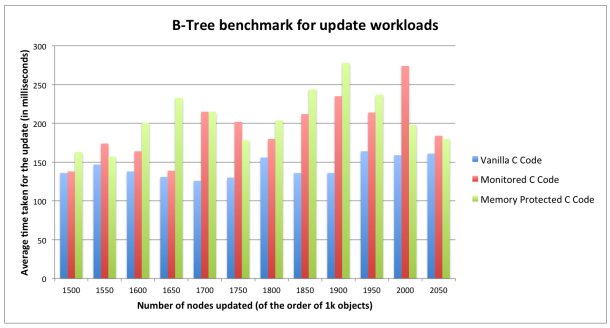

Figure 4: Binary Tree Benchmark Results For Update Workloads

dom objects from a Binary Tree of a fixed size (number of nodes in the binary tree was kept to 2500). For the read tests, we generate a random set of keys and search for 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, while the page protection mechanism has an overhead of 28% (refer to Fig. 3). Tracer incurs slightly more overhead because it performs fine grained monitoring, whereas page protection mechanisms provide an approximation of the access pattern of the application. These overheads are reasonable considering the more precise information that Tracer provides the developer with.

Binary Tree Update Tests The third set of tests measure the overall overhead incurred by update intensive workloads. 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 key's value. If the source key is not found in the binary tree then the update procedure returns. We varied 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 the page protected mechanism by 6.28%. The overhead over the vanilla C implementation is around 35% (refer to Fig. 4).

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

An important observation from tree traversal and creation tests is that the 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 very 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 the application developer with accurate information about the application's memory access pattern. This information could be used for 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 runtime) which would move and pack objects with similar access patterns together in the same pages. The dynamic memory management system 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 the 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 the 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 the 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 an application design, from which Tracer would provide fine grained memory access information. The developer could then enhance the application's performance by adequately restructuring the application's memory usage (for example, by allocating frequently accessed objects together).

Improvement in interceptor's capabilities Enhancements could be made to the parsing abilities of Tracer's interceptor. At the moment, the subset of C which the parser supports focuses on structs and simple C syntax. In

order to support more idiomatic C, richer features would have to be added. Primary amongst these is support for arrays, which is likely to require some careful implementation. Given the current header-based counting used in Tracer, there are several considerations for array suppport. The header can be added per-array (perhaps for primitive C types), or per-object. To keep per-object tracking may require modifications to the instrumented code, such as adding the counter as a struct member, rather than as a header. For arrays of pointers, the counts would likely be unnecessary.

6 Related Work

Memory tracing mechanisms such as Valgrind's memcheck [8], gdb's ptrace [1], SSDAlloc's [2] page protection mechanism rely on system call interfaces for fine grained memory access. The overheads incurred due the reliance on system calls can downgrade the performance of the application. Kernel crossings result in interrupts to kernel, mode switches and data and instruction cache pollution. *Indirection* mechanisms (such as those based on handles) have been previously used in HAC [4] and Mac memory management systems. Handle, based approaches trade off backward compatibility and provide finer monitoring of application data. We next describe systems based on memory protection mechanisms.

SSDAlloc's page protection mechanism SSDAlloc monitors fine grained access patterns through the use of an 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. Workloads that touch larger memory segments (such as tree creation, tree traversal) may not be fully resident within the page buffer. This could lead to heavy overheads due to interrupt call, object eviction from page buffer and page materialization.

Chameleon's memory tiering mechanism Chameleon 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. Chameleon requires memory object alignment at boundary offsets. This could lead to some memory wastage over standard memory allocation schemes. As with SSDAlloc the OPP model increases TLB pressure. Tracer does not make such assumptions for tracking object accesses.

Memory tracing systems Our design is inspired from *memlet* based systems []. *Memlets* are small pieces of code which are injected into application code. These code sequences execute additional code for every memory access and monitor application access pattern at a finer granularity. Memlets are generally weaved into into application code through *binary translation techniques, watchpoints, taint checking and data flow analysis*. Binary translation techniques modify code after the the application has been compiled to binary format. Binary translators [3, 8]convert programs to intermediate representation and then perform dynamic or static translation to improve the overall performance. Greathouse et al. [5] describe a system for adding unlimited *watchpoints* using hardware extensions (bitmap translation look aside buffer and a range cache) which can provide fine grained memory monitoring. Taint checking [6, 9] and data flow analysis systems use additional taints per memory location and registers for registering accesses to them. Tracer "taints" memory structures with metadata (as headers) for monitoring accessing and updating access count information.

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 headers transparently and provides interfaces to application developers for getting useful information about memory access patterns. We also implement protection based mechanism based on SSDAlloc [2] and perform microbenchmark studies using binary trees. Our implementation of a prototype of Tracer shows reasonable overheads of 28% and 40% over non-modified C code for creation and read heavy workloads, respectivley. In comparison to page protection mechanism, Tracer improves the average performance by 12% and 6.28% for read and update heavy workloads, respectively.

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] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[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] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. *ACM SIGARCH Computer Architecture News*, 40(1):159–172, 2012.

[6] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 29–41. ACM, 2006.

[7] C. H. Lam. Storage class memory. In *Solid-State and Integrated Circuit Technology (ICSICT), 2010 10th IEEE International Conference on*, pages 1080–1083. IEEE, 2010.

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

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

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