Implementing Stack Simulation for Highly-Associative Memories[†]

Yul H. Kim, Mark D. Hill, David A. Wood

Computer Sciences Dept., 1210 West Dayton St.
Univ. of Wisconsin, Madison, WI 53706

ABSTRACT

Prior to this work, all implementations of stack simulation [MGS70] required more than linear time to process an address trace. In particular these implementations are often slow for highly-associative memories and traces with poor locality, as can be found in simulations of file systems. We describe a new implementation of stack simulation where the referenced block and its stack distance are found using a hash table rather than by traversing the stack. The key to this implementation is that designers are rarely interested in a continuum of memory sizes, but instead desire metrics for only a discrete set of alternatives (e.g., powers of two). Our experimental evaluation shows the run-time of the new implementation to be linear in address trace length and independent of trace locality. Kim, et al., [KHW91] present the results of this research in more detail.

KEY WORDS: trace-driven simulation, stack simulation, caches, memory systems, file systems.

1. Introduction

Trace-driven simulation, the most commonly-used method for evaluating cache and memory system designs, can consume considerable computer resources, particularly when it is applied to highly-associative memories with traces having poor locality. A key approach to reducing these resource demands is to evaluate multiple alternative memories with a single pass through an address trace.

Mattson, et al., [MGS70] describe a single-pass technique called *stack simulation* that can efficiently simulate multiple alternative memories with the same block size (line, page), the same number of sets, and using the least-recently-used (LRU) replacement policy. Since stack simulation handles each set independently, we will hereafter assume fully-associative memories (a single set). Mattson, et al., show why a single list, called a *stack*, can be used to simultaneously represent the contents of all alternative memories, and how the first k elements of the stack give the blocks in a memory of size k blocks. In stack simulation, a reference is said to be to *stack distance* k if it accesses the k-th block in the stack. Each reference is processed in four steps:

INPUT read the next reference from the trace,

FIND search the stack to find the block the reference accesses (if any) and determine its stack distance,

METRIC update counter(s) to record which memories hit or missed, and

UPDATE update the stack to reflect the state of the memories after the reference.

A straight-forward implementation of stack simulation uses a linked list and has asymptotic running time $O(N \cdot D)$, where N is the trace length and D is the mean stack distance. This implementation is commonly-used for CPU cache simulations, where the limited associativity restricts D and traces exhibit good locality [Tho87]. Others have developed tree-based implementations [BeK75, Olk81] with asymptotic running times $O(N \log D)$ and used hashing to determine when a block is *not* in the stack [BeK75, Tho87]. The performance of these implementations, however, is still sensitive to trace locality [Tho87].

In the next section we develop an implementation that uses hashing to achieve asymptotic running times that are independent of trace locality. We then present key results from our experimental evaluation.

2. Hashing-Based Implementation

Our implementation of stack simulation uses a hash table to find a reference in the stack. This implementation would be trivial, except that step FIND must also determine the block's stack distance in constant time. One way to achieve this is to add a field to each block that stores its stack distance. Doing this, however, would not improve asymptotic running time, because step UPDATE would now require an average of D updates per reference.

The key to our implementation is that designers are rarely interested in a continuum of memory sizes, but instead desire metrics for only a discrete set of alternatives (e.g., powers of two). Thus, we need only remember a block's stack distance at that coarse level, and therefore we

[†] This work is supported in part by the National Science Foundation (MIPS-8957278 and CCR-8902536), A.T.& T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation.

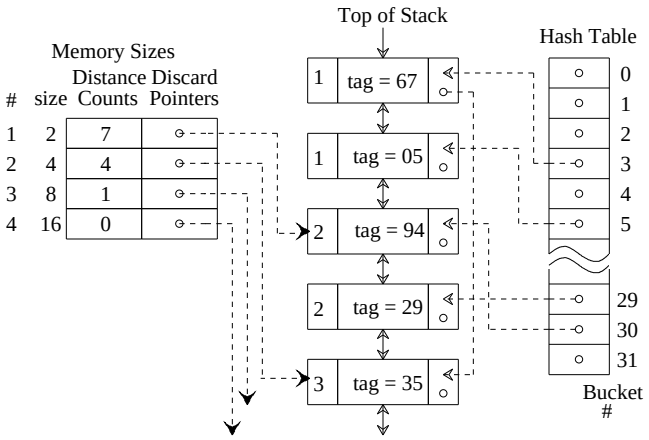

Figure 1a: Example of stack and hash table.

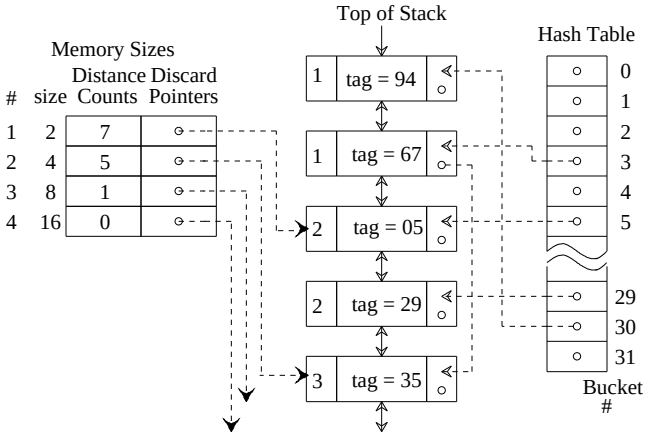

Figure 1b: Same stack after reference to 94.

must update the field only when a block moves from one memory to the next. Let the number of memory sizes being simulated be M. Our implementation has asymptotic running time $O(N \cdot M)$, because on each reference a single memory size requires at most one update. The expected number of updates per reference, however, is equal to the sum of the M miss ratios. Since miss ratios for highly-associative memories are typically small, our implementation is fast unless M is very large.

Figure 1 shows the key data structures of our implementation, simulating (very small) memory sizes of 2, 4, 8 and 16 blocks. The stack is implemented with a doubly-linked list, hash table, and discard pointers (which point to the first block *not* in a memory)[†]. Figure 1a shows the stack before processing a reference to block 94, while Figure 1b shows it afterward.

[†] Robinson and Devarakonda [RoD90] use similar data structures to implement frequency-based replacement in a disk cache.

Using the simulation steps defined in Section 1, a reference to block 94 is processed as follows.

INPUT ‘‘94’’ is read from the trace.

FIND Its block is found by hashing (with 94 mod 32) through bucket 30.

METRIC Since the ‘‘distance’’ field for the block says ‘‘2’’, the distance count for memory 2 must be incremented to indicate a hit in memories 2 and larger.

UPDATE Finally, the discard pointer for memory 1 (all memories less than 2) must be moved up one block, the discarded block marked out of memory 1, and block 94 moved to the top of the stack.

3. Discussion

We have implemented hashing-based and linked-list stack simulation in C and have evaluated them with some CPU address traces [BKW90]. We synthesized various mean stack distances using stack deletion [Smi77]. Results show that the performance of hashing-based stack simulation is (1) insensitive to mean stack distance, and (2) faster than linked-list stack simulation for mean stack distances greater than five. Thus our implementation should be superior for disk and file-system traces, which have mean stack distances in the hundreds [Tho87].

4. References

[BeK75] B. T. Bennett and V. J. Kruskal, LRU Stack Processing, *IBM Journal of R & D*, July 1975, 353-357.

[BKW90] A. Borg, R. E. Kessler and D. W. Wall, Generation and Analysis of Very Long Address Traces, *Proceedings Seventeenth International Symposium on Computer Architecture*, Seattle, June 1990.

[KHW91] Y. H. Kim, M. D. Hill and D. A. Wood, Implementing Stack Simulation for Highly-Associative Memories, Computer Sciences Technical Report #997, Univ. of Wisconsin, February 1991.

[MGS70] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems Journal 9*, 2 (1970), 78 - 117.

[Olk81] F. Olken, Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies, Masters Report, Lawrence Berkeley Laboratory LBL-12370, University of California, Berkeley, May 1981.

[RoD90] J. T. Robinson and M. V. Devarakonda, Data Cache Management Using Frequency-Based Replacement, *Proceedings SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, Colorado, May 1990.

[Smi77] A. J. Smith, Two Methods for the Efficient Analysis of Memory Address Trace Data, *IEEE Trans. on Software Eng. SE-3*, 1 (January 1977), 94-101.

[Tho87] J. G. Thompson, Efficient Analysis of Caching Systems, Computer Science Division Technical Report UCB/Computer Science Dept. 87/374, University of California, Berkeley, October 1987.