IMPLEMENTING STACK SIMULATION FOR HIGHLY-ASSOCIATIVE MEMORIES

by

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

Computer Sciences Technical Report #997

February 1991

Implementing Stack Simulation for Highly-Associative Memories[†]

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

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. This allows the trace-driven simulation of multiple alternative memories with the same block size, the same number of sets (e.g., fully associative), and using the least-recently-used replacement policy, with one pass through the trace in linear time. The key to this implementation is that designers are rarely interested in a continuum of memory sizes, but instead desire metrics for only a small, discrete set of alternatives (e.g., powers of two). We determine the memories in which a block resides by augmenting the state of each block with an index to the largest memory that contains that block. We update this state by using pointers to the block below the least-recently-used block in each memory. Our experimental evaluation confirms that the run-time of the new implementation is linear in address trace length and independent of trace locality.

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

1. Introduction

Trace-driven simulation is the most common method for evaluating cache and memory system designs. A major shortcoming is the hours of execution time a single simulation can consume, particularly when it is applied to highly-associative memories with traces having poor locality. Traces of billions of references are being used today [BKW90], which only exacerbates the problem. Simple and efficient algorithms are needed to reduce simulation time. A key approach is to evaluate multiple alternative memories with a single pass through an address trace. But while good algorithms exist for direct-mapped and set-associative caches [HiS89], the algorithms for fully-associative memories are less attractive.

Mattson et al. [MGS70] describe a single-pass technique called *stack simulation* that can efficiently simulate multiple memories with the same block size (line, page), the same number of sets, and using the least-recently-used (LRU) replacement policy. Stack simulation takes advantage of *inclusion*, the property that a larger cache or memory always contains a superset of the blocks (or pages) contained in smaller ones. A single stack can therefore simultaneously represent the contents of many alternative memories, with each memory of size k blocks consisting of the top k blocks of the stack. Any reference to a block at level k in the stack, called the reference's *stack distance*, therefore hits in all alternative memories of size k blocks or larger. Simulations maintain *distance counts* that record the number of references found at each stack distance. Summing the distance counts for depths one to k gives the number of hits to a memory of size k blocks; the miss ratio is easily derived from that.

[†] The material presented here is based on research supported in part by the National Science Foundation's Presidential Young Investigator and Computer and Computation Research Programs under grants MIPS-8957278 and CCR-8902536, A. T. & T. Bell Laboratories, Cray Research, Digital Equipment Corporation, and the graduate school at the University of Wisconsin-Madison.

Linked lists provide a straight-forward implementation of LRU stack simulations. Since finding references in the stack (and therefore the reference's stack distance) involves traversing the linked list, these implementations have running times proportional to the length of the trace times the mean stack distance [Tho87]. Bennett and Kruskal [BeK75] and Thompson [Tho87] use hash tables to supplement their linked-list stacks. The tables determine if a reference is not contained in the stack, in which case a futile search of the stack can be avoided. However, traversal of the stack is still required to determine references' stack distances, such that these algorithms still run in time proportional to the length of the trace times the mean stack distance [Tho87]. CPU traces usually exhibit good locality, so CPU cache simulations usually have small mean stack distances (even with fully-associative caches); Thompson reports mean stack distances between seven and twenty [Tho87]. For implementation reasons, CPU caches typically limit associativities to at most four, further restricting the mean stack distance. Thus simple linked-list implementations give good performance for CPU cache simulation.

However, simulations are also done for file systems [ODH85] and disk caches which are generally fully-associative and exhibit much higher mean stack distances; Thompson [Tho87] observed mean stack distances between 200 and 500 for disk and file system traces. Filtered input traces, used to reduce simulation time, may also experience large mean stack distances. One filtering technique is *stack deletion*, in which references that would otherwise hit near the top of stack are deleted [Smi77] (see Section 3.1 for a more complete description). Stack deletion increases the mean stack distance by greatly reducing the locality in a trace. Simulations for large virtual memories, file systems, or that use traces reduced through stack deletion will therefore have poor running times with linked-list stack simulations due to their large mean stack distances.

Bennett and Kruskal [BeK75] and Olken [Olk81] have proposed LRU simulation algorithms that encode the stack's state in a tree. These methods reduce the number of elements that must be searched to determine a reference's stack distance (or determine that the reference is not in the stack). Bennett and Kruskal's algorithm takes advantage of the fact that a block's stack distance is just the number of unique blocks referenced since the block's last reference [Tho87]. Simulation times become proportional to the log of the average inter-reference time. Olken further refines Bennett and Kruskal's algorithm to run in bounded space where the alternative memories are smaller than some given size. Olken develops his own algorithm that uses an AVL tree to encode the state of the stack. The tree is dynamically rebalanced, and thus exhibits a better worst-case running time than Bennett and Kruskal's algorithm. Thompson shows that further improvements, to maintain a better balanced tree with fewer rebalancing actions, reduce the algorithm's complexity [Tho87]. However, Thompson's measurements indicate that Bennett and Kruskal's algorithm typically gives the best running time of the tree-based methods. In both cases, Thompson has shown that these tree-based algorithms reduce the asymptotic running time to $O(N \log D)$, where D is the mean stack distance.

Unfortunately, the tree-based schemes lack the clean intuitive appeal of linked-list stack simulation and are much more difficult to code. They also have running times that are still related to the locality present in the trace, and thus will have worse performance for traces that induce high mean stack distances (albeit varying according to the log of the mean stack distance or inter-reference time). This paper shows that our hashing stack algorithm has run times unrelated to a trace's locality, resulting in run times proportional to trace length.

Trace-driven simulation and execution-time profiling confirm that our hashing stack algorithm can process each reference in approximately constant time. However, the overhead of maintaining the hash table results in a large constant factor, causing linked-list-based simulations to perform faster for traces with low mean stack distances. The hashing algorithm overcomes the constant overhead and clearly outperforms linked-list stack algorithms for mean stack distances greater than 10.

The rest of this paper is organized as follows. Section 2 presents our hashing stack algorithm, and also presents an informal analysis of its running time. Section 3 compares the execution times of a simulator using the hashing algorithm with those of a linked-list-based simulator. Section 4 discusses possible extensions to and limitations of the hashing algorithm. Finally, Section 5 presents conclusions.

2. A Hashing Stack Algorithm

Our hashing stack algorithm can simulate multiple memories with one pass through the trace so long as all memories use the same block size, the same number of sets (one for fully-associative), and LRU replacement. Like conventional stack algorithms, the hashing stack algorithm represents the stack using a linked list. The important difference is the addition of a hash table to accelerate the lookup function. The hash table serves as an index, returning a pointer to the correct stack element (or NIL) in essentially constant time. However, merely locating a stack element is insufficient; we must determine its current stack depth.

The key to the algorithm 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 the algorithm need only maintain distance counts for these sizes: a hit at distance n is recorded in the distance count of the smallest memory containing at least n blocks. We can efficiently compute distance counts if each stack element has an *in-memory* field, that keeps track of the smallest memory that contains it. In the remainder of this section, we describe the algorithm in detail and examine its run-time complexity. The algorithm consists of four stages for each reference: read in the reference, locate the reference in the stack (or determine that the reference is not in the stack) and determine its stack distance, update metrics to indicate which memories contain the reference, and then update the stack to reflect changes in the contents of the memories after this reference [HiS89]. We call these stages INPUT, FIND, METRIC, and UPDATE, respectively. After first describing the data structures, we examine each of the four stages in turn.

2.1. Data Structures

A doubly-linked list represents the state of the stack, with each element of the stack corresponding to a unique block encountered in the reference stream (see Figure 2). The top stack element, or block, is most recently referenced, and so on. Addresses are hashed by block number (the integer address divided by the block size in bytes). Each non-empty hash bucket points into the doubly-linked stack to the block in which the reference occurred, and uses chaining to resolve collisions [Knu73]. In our implementation, the hash table entries are combined with the stack nodes to simplify storage management. The hash function (bucket# = reference-block# MOD hash-table-size) is simple and fast, while causing few collisions.

We chose the size of the hash table rather arbitrarily to be 8201 or 10001 buckets. Clearly, power of two table sizes will speed up the hash function (which reduces to bit selection), but may also cause more collisions. We made no efforts to examine this tradeoff or refine the hash function, since we observed a very low collision rate with the present scheme. An execution profile indicates that the hash function consumes less than 6% of the simulation time in all cases (see Section 3.2 for results indicating low collision rates).

Each stack element contains a field that indicates the largest memory size of interest in which the block currently resides, called the *in-memory* field. A separate array of pointers, one for each simulated memory size, helps maintain the in-memory fields. The *discard-pointer* for a memory size of N blocks points to the block at stack distance $N+1$.[1] We also keep track of the size of the stack in a variable called *stack_size*. See Figure 1 for an illustration of a stack element, and Figure 2 for an example of a stack, hash table, and associated data structures.[2]

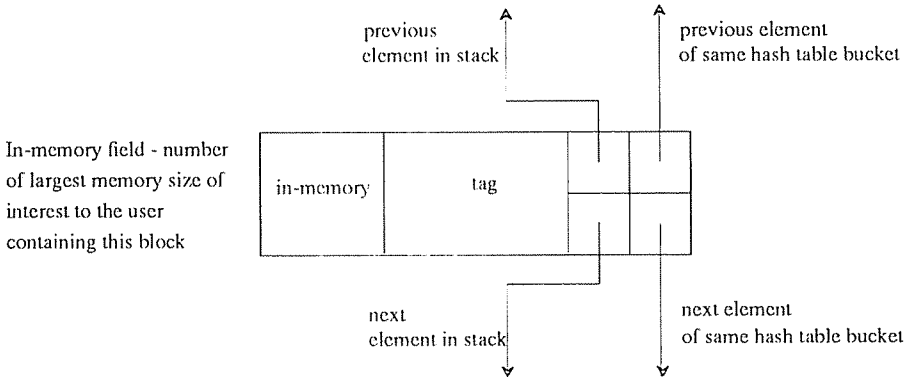

Figure 1: Stack element.

[1] Robinson and Devarakonda [RoD90] use a similar set of pointers indexing into an LRU stack to implement frequency-based replacement in a disk cache.

[2] In our implementation of the algorithm, we added a backpointer per element to make pruning in UPDATE more convenient.

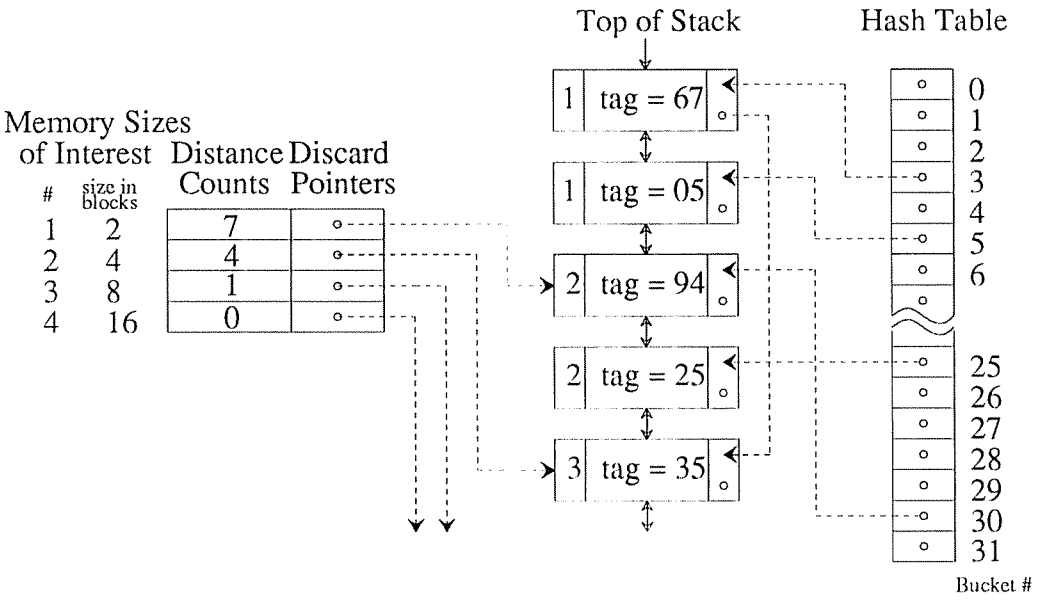

Figure 2: Example of the stack, hash table and associated data structures.

This example shows four memory sizes of interest consisting of two, four, eight, and sixteen blocks, respectively. A memory of size two contains the top two blocks of the stack (67, 05), of size four the top four blocks (67, 05, 94, 25), etc. Note that there is a discard-pointer (to the next block after the last one in each size) and distance count associated with each size. The hash table shown has 32 buckets; the hash function is the block's tag MOD the table size (hence why 67 and 35 are linked to the same bucket.)

2.2. INPUT

INPUT reads in references in ASCII as 4 byte hexadecimal addresses. We used ASCII to aid with debugging despite the obvious speed disadvantages. Our algorithm as presented here makes no distinction between instruction and data references, or reads and writes.

2.3. FIND

To find a reference's block in the stack, we hash the address of the reference to obtain a bucket number in the hash table. If the bucket is empty, the reference is not in the stack. If it is non-empty, we follow the bucket's collision chain, comparing the tag of the reference's block with those of each block in the chain. On a match, we return a pointer to the correct entry. If the entry is not in the collision chain, it is not in the stack and the reference misses. The average number of elements that must be searched to find a block in the hash table is approximately $1 + \frac{\alpha}{2}$ where α equals the number of blocks hashed into the table divided by the number of buckets in the table [HoS76]. For a sufficiently large hash table, FIND can be done in essentially constant time.

FIND, in addition to finding a block in the stack, must determine its stack distance. If the block is found, its in-memory field directly indicates the largest memory size which contains it (i.e. its stack distance), with no traversal of the stack.

2.4. METRIC

At this point we have either found the reference's block or determined that the block is not in the stack. If the block is found, we use its in-memory field to directly increment the proper distance count to reflect a hit in that and all larger memory sizes (see Figure 3). Note that we do not require traversal of the stack to compute stack distances, and thus update the appropriate metric for each reference in constant time.

If a block is not found in the stack, METRIC records it as a miss in all memories. METRIC also increments stack_size, since misses cause the stack to grow in size.

2.5. UPDATE

The algorithm then updates the stack. As in conventional stack algorithms, the reference's block moves to the top of stack, if it is not already there (see Figure 4). If it was originally at stack distance *d*, the blocks at distances 1 through *d* - *1* automatically move down one block (the whole stack moves on a miss). The hashing stack algorithm must also update the appropriate discard-pointers and in-memory fields. We move discard-pointers that index into the stack at or above the depth of the reference (the size of the stack on a miss) up to point to the next higher block in the stack (the new block just below the last block of that size of interest - it is for this reason that the stack must be doubly linked). We also increment the in-memory fields of these new blocks (having just been pushed out of the next smaller memory size). At the start of a simulation, the stack grows as blocks are referenced for the first time. Stack_size indicates when to initialize discard-pointers as each successive memory size of interest fills up. Note that if a block is found in the stack, UPDATE does not affect any portion of the stack or its associated data structures below the distance of the reference.

Blocks that fall out of the largest size of interest are pruned out of the stack and hash table (at the same time we also decrement stack_size). This does not affect metrics derived from the simulation since any block found below the last size of interest in the stack is a miss anyway, and it allows the simulation to run for arbitrarily long traces.[3]

[3] In our implementation of the algorithm, we actually retain one block beyond the end of the last size of interest, as it is pointed to by the discard-pointer of the largest memory.

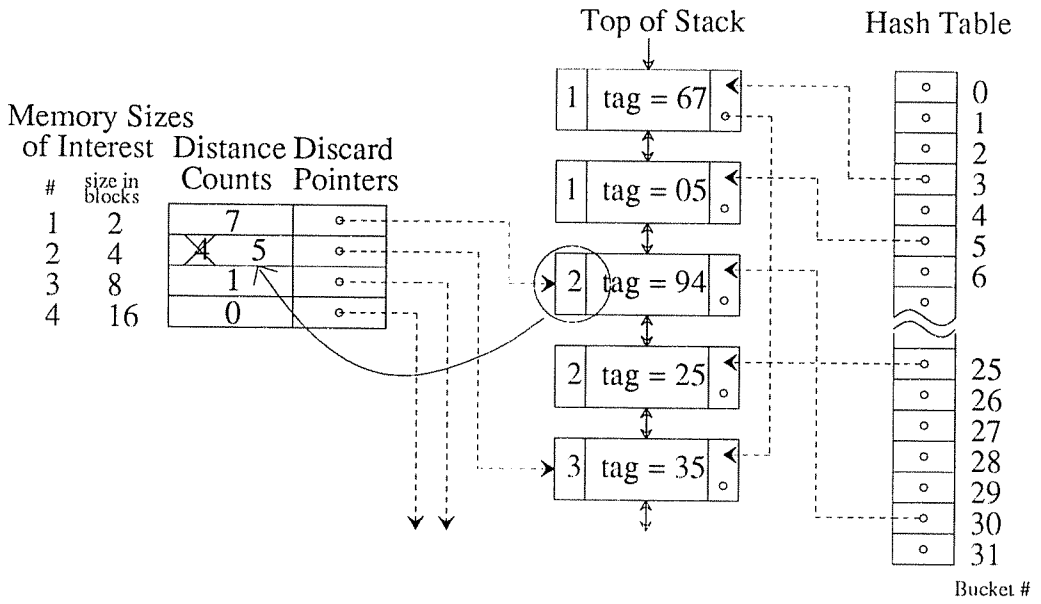

Figure 3: This and the next figure show the same stack and hash table while processing a reference to block 94. Here we have just done FIND and METRIC. The tag hashed into the table (94 MOD 32 = bucket #30) gives a pointer directly to the correct block. Note that the in-memory field of that block gives its stack distance without traversing the stack, indicating that the second distance count must be incremented, from 4 to 5.

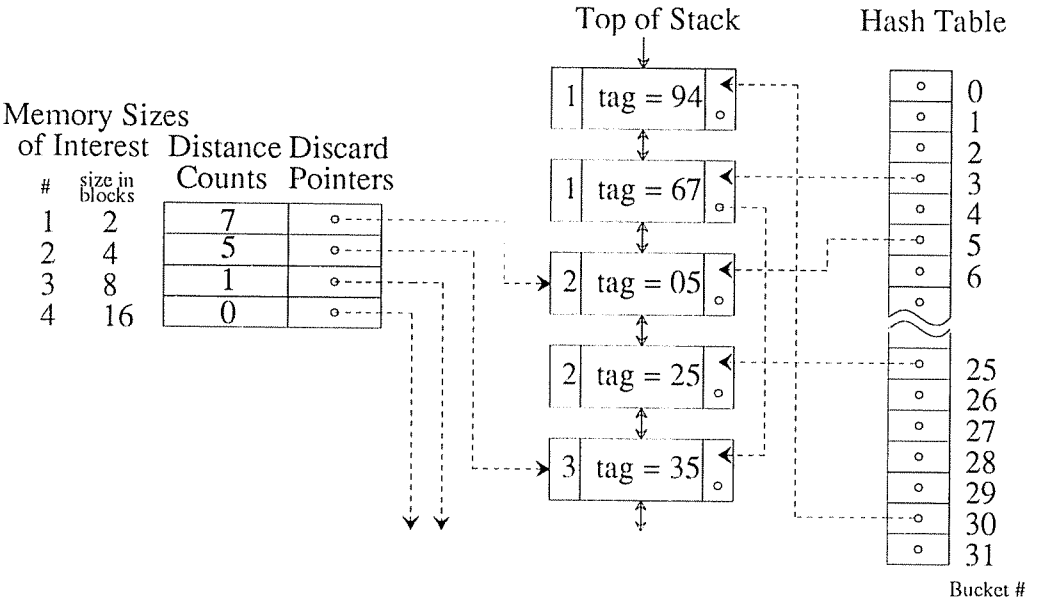

Figure 4: In doing UPDATE, block 94 is moved to the top of stack (LRU) and its in-memory field changed. All blocks above the depth of the hit move down automatically. Discard-pointers in this range are moved up (there is only one - discard-pointer #1 now points to block 05) and the in-memory fields of the new blocks they point to are incremented. Note that updating takes place in the stack only above the depth of a reference (the depth of a miss can be thought of as equal to the size of the entire stack).

To determine UPDATE's time per reference, let us assume that the simulation is in steady-state such that all M memory sizes are full, and that the miss ratio for the i th size is m_i, for i = 1..M. The amount of time it takes to move a block to the top of stack is constant; call this A. The amount of time it takes to adjust a single discard-pointer and the in-memory field of the block it points to is also constant; call this B. For each reference we must then spend an amount of time equal to A to move the block to the top of stack, plus B for each memory whose size (in blocks) is smaller than the stack distance of that reference (or in other words, for each memory size for which this reference is a miss). Thus UPDATE requires $A + B \sum_{i=1}^{i=M} m_i$ time per reference, on average, which is always less than or equal to $A + \left\lceil B \times M \right\rceil$ since miss ratios are never greater than one. In fact, these miss ratios are likely to be quite small. Regardless, the time per reference to do UPDATE is $O(M)$, where M is unrelated to stack size, trace length, or other parameters, and tends to be a small constant (less than eight).

2.6. Comparison to Other Algorithms

We have shown that our algorithm does INPUT, FIND, METRIC, and UPDATE in constant time. (The validity of assuming constant time hash table operations for the FIND operation will be examined experimentally in Section 3.2.) In total, our algorithm processes a trace of length N in at most $O(NM)$ time, where M is a small constant chosen by the designer. Linked-list based algorithms have run times of $O(ND)$, and tree-based algorithms asymptotic run times of $O(N \log D)$, where D is the mean stack distance. For traces with high mean stack distances, D is large while M is always small and unrelated to the locality of input traces in any way.

3. Evaluation

We implemented the hashing stack algorithm described above and compared it to a conventional linked-list LRU implementation. We ran both simulators on a variety of input traces to compare their run times, and to test the assertion that the hashing algorithm would run in time proportional to trace length (i.e. is insensitive to locality).

3.1. Traces

The input traces were selected portions of long CPU address traces generated through link-time code modification techniques [BKW90]. We would have preferred to use file system traces, but had none at our disposal. Tables 1 lists the names and characteristics of each trace.

Name	Description	Length (10^6 instrs.)	Mean Stack Distance
Mult1	Multiprocessor traces from 6 processes running concurrently.	127	3.2
Mult2	Multiprocessor traces from 6 processes running concurrently.	137	3.0
Tree	Lisp dialect program that builds a tree and then searches for the largest element. Does garbage collection when needed.	157	2.8
Tv	A timing verifier for VLSI circuits, written in Pascal. Generates then traverses a linked list structure.	195	12
SOR	Fortran implementation of a successive overrelaxation algorithm using sparse matrices.	97	52

Table 1: Characteristics of Five Main Traces

This table describes the five main traces used in simulation. Lengths are in millions of instructions. Mean stack distances are those that would have been obtained for a fully-associative LRU stack of unbounded size with 1K byte blocks, where the distance of misses are recorded as the size of the stack at the time of the reference.

Derived From:	Deletion Depth	Length (10^6 instrs.)	Mean Stack Distance	Relative Error 1M memory
Mult1	1	38	8.5	0%
	2	23	13	0%
	4	9.3	28	0.01%
	8	4.4	53	0.02%
	16	1.8	120	0.14%
Mult2	1	41	7.7	0%
	2	25	11	0%
	4	9.1	26	0.01%
	8	4.7	47	0.05%
	16	2.2	89	0.15%
Tree	1	49	6.6	0%
	2	34	8.7	0%
	4	14	17	0%
	8	6.9	27	0%
	16	2.3	55	0%
TV	1	75	29	0%
	2	43	48	0%
	4	17	120	0%
	8	6.3	310	0%
	16	6.2	320	0%
SOR	1	20	260	0%
	2	17	310	0%
	4	9.3	550	0%
	8	4.0	1200	-.02%
	16	3.7	1400	-.04%

Table 2: Characteristics of Other Traces

These traces were obtained through the stack deletion. Note the increase in mean stack distance associated with larger deletion depths. The last column shows the relative error in results obtained with these traces versus those obtained with the original traces, for fully associative LRU simulations of 1M memories with 1K blocks. 0% indicates that $-0.005\% \leq error < 0.005\%$.

Table 2 lists additional traces derived from the original five using stack deletion, a reduction technique that deletes from the trace all references that hit within the top d blocks of an LRU stack [Smi77]. The parameter d is called the *deletion depth*. Besides reducing trace length, stack deletion also reduces the trace's locality, and thus increases the mean stack distance that the trace would induce in a stack simulation. From here on, it will be more convenient to associate mean stack distances with the traces themselves, and unless otherwise noted it will refer to the mean stack distance that would be obtained for a fully-

associative LRU stack of 1K byte blocks which is allowed to grow to an arbitrary size (stacks which are bounded in size due to limited associativity or for efficiency considerations will decrease the mean stack distance). The depths of misses are recorded as the size of the stack at the time of the reference.

Stack deletion is an approximate technique, and can introduce error by skewing the simulation's distance counts and miss ratios [Smi77]. However, in this paper we are interested primarily in our simulator's efficiency rather than its results. Stack deletion gives us a convenient way to vary the amount of locality and mean stack distance of the traces used in our evaluation. In practice, file system traces have mean stack distances much larger than those of CPU traces; Thompson reports CPU traces with mean stack distances in the range of 7.5 to 71 and disk and file-system traces in the range of 240 to 500, not including misses [Tho87]. The last column of Table 2 lists the relative error caused by stack deletion for a 1M fully-associative memory with a 1K byte block size.

It should also be noted that the link-time modification techniques used to generate the five main traces lose some information about the interleaving of instruction fetches and data and stack accesses within basic blocks of instructions [BKW90]. As a result, these main traces may exhibit more locality and thus lower mean stack distances than might actually observed from the unaltered reference stream as seen by a unified memory. Once again, we do not care if the range of mean stack distances of our traces may start somewhat low since we are studying our simulator's efficiency (3/4 of Thompson's CPU traces have mean stack distances above 12 even though he excludes references to previously unreferenced blocks).

3.2. Methods and Results

We used the traces listed above as input to both simulators. We wrote the simulators in C, compiled them with the highest level of optimization, and ran them on a MicroVAX 3200[4] with some X11[5] processes in the background. Virtual run times were measured by calls to *getrusage*. We did separate runs for profiling purposes for use with the UNIX[6] profiler *gprof*. In all cases, we simulated fully-associative memories of sizes 256K, 512K, 768K, 1M, 1256K, 1512K, and 2M, with a block size of 1K bytes. Although 1K byte blocks may seem inappropriate for CPU caches, as well as disk caches and file systems, we made our choice for the following reasons. Previous sections of the paper argue especially for the use of hashing stack algorithms with disk and file system traces. However, we did not have any at our disposal; nor did we have CPU traces long enough to simulate memories with sizes on the order of file or disk

[4] MicroVAX 3200 is a registered trademark of the Digital Equipment Corporation.
[5] X11 is a registered trademark of the Massachusetts Institute of Technology.
[6] UNIX is a registered trademark of AT&T.

systems. In order to give an idea of how our hashing algorithm would perform in this domain, we tested both simulators with the CPU traces described above, with memory and block sizes scaled down appropriately (i.e. 256K to 2M memories with 1K byte blocks).

Figure 5 shows the time per reference versus mean stack distance. Each graph, except the last, presents the results for a single base trace together with its derivative traces. Note that the time per reference remains fairly constant for the hashing algorithm, but varies proportional to mean stack distance for the linked-list implementation. The last graph in Figure 5 combines the results from all five trace groups to illustrate the qualitative similarities between the five sets of results. Note that the hashing lines are all more or less horizontal and that the linked-list lines all have the same trend.

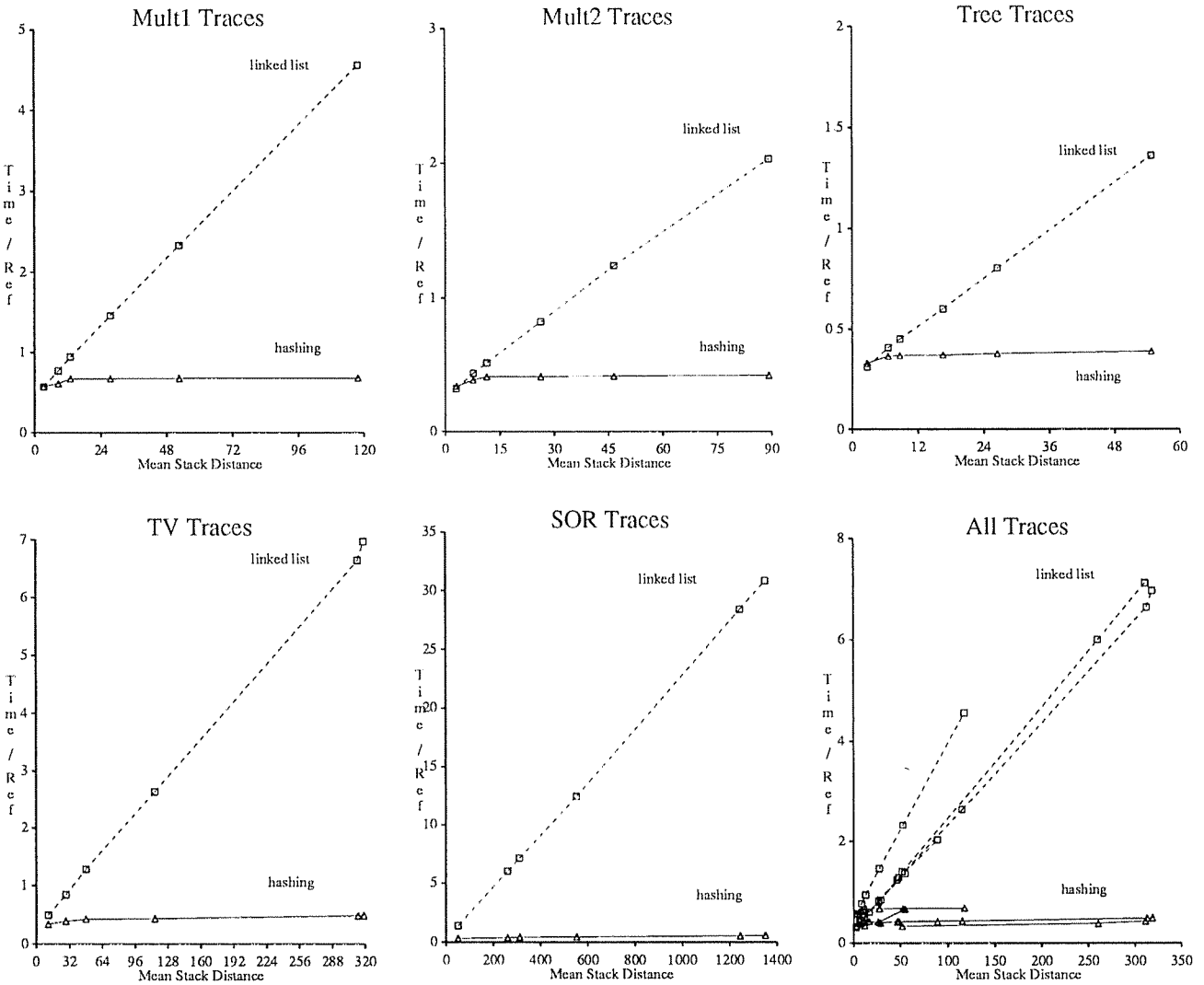

Figure 5: Time/ref vs. Mean Stack Distance for Linked List and Hashing

These graphs show time/ref (virtual msec.) for both the linked list and hashing simulations vs. the mean stack distance for the traces used. All five traces sets are displayed separately, and then together in the last graph for qualitative comparison (SOR runs with mean stack distances above 320 are not shown on the last graph). Note the different scales on all graphs.

The graphs clearly show that hashing is superior for large mean stack distances. However, if mean stack distances are low, constant factors may dominate the simulation time. The extra constant overhead needed to maintain the doubly-linked list, discard-pointers and hash table result in shorter run times for the linked-list algorithm in some cases. Out of the five base traces, the linked-list implementation performs better for the two multiprogramming workloads and Tree. The cross-overs occur at mean stack distances ranging from approximately 3.5 to 4.3. As the mean stack distances increase, the constant factors become small compared to searching the linked-list, and hashing delivers better performance for the other base traces and all reduced traces.

The UNIX execution profiler gprof confirms the tradeoff of higher pointer manipulation overhead for faster search time. Tables 3 and 4 present an approximate breakdown of time spent doing INPUT, FIND, METRIC, and UPDATE for simulations using traces derived from Mult2. Two statistics are presented, the percentage of time spent in each stage and the average time per reference. The tables show that both simulators spend roughly equal time in METRIC and INPUT. The hashing algorithm spends roughly constant time per reference in FIND, while the linked-list implementation spends time proportional to the mean stack distance. The extra overhead required by the hashing algorithm causes it to spend roughly twice as much time per reference performing UPDATE. Note that the hashing algorithm does spend more time in UPDATE as the mean stack distance increases. This occurs because UPDATE must modify more discard-pointers and in-memory fields for larger stack distances. However, the time in UPDATE is bounded since we are interested in a fixed number of memory sizes. In addition, the magnitude of the increase is very small: less than 4% over the entire range of mean stack distances.

Our simulator used a simple hash function, block number modulo table size, and a fixed table size. Further tuning of the hash function and table size could further reduce the average number of hash probes and thus the the run-time of FIND. However, the average number of tag comparisons over all simulations is just 1.05, and in all cases is less than 1.23. Gprof also shows that the simulator never spent more than 6% of its time doing division and remaindering for the hash function.

The time complexity analyses of previous stack algorithms usually ignore INPUT, which is reasonable since those algorithms all grow faster than $O(N)$. Note that for our hashing algorithm, INPUT takes a significant portion of time (due to the ASCII input format), which if reduced, makes the advantages of constant time FIND even more significant.

The memory requirements of our algorithm are quite modest. Pruning the stack and hash table limits their size to be proportional to the largest memory size of interest. Each stack element requires only 16 extra bytes to accommodate the in-memory field, stack back pointer, and double collision chain pointers. In

Deletion Depth	Mean Stack Distance	INPUT		FIND		METRIC		UPDATE	
		%	time/ref	%	time/ref	%	time/ref	%	time/ref
0	3.0	48	.15	30	.09	6.5	.02	7.1	.022
1	7.7	45	.20	40	.18	4.5	.02	6.1	.027
2	11	38	.20	49	.26	3.8	.02	5.0	.026
4	26	25	.20	66	.53	2.6	.02	3.4	.027
8	47	17	.20	77	.90	1.8	.02	2.4	.028
16	89	11	.20	85	1.6	1.1	.02	1.5	.029

Table 3: Profiling of Linked-List Implementation, Mult2 Traces

Table 3 shows the breakdown of Mult2 run times done with gprof for the linked-list implementation. % indicates the total amount of time spent in that activity as a percentage of the total time to process entire trace. Time/ref is in virtual msec.

Deletion Depth	Mean Stack Distance	INPUT		FIND		METRIC		UPDATE	
		%	time/ref	%	time/ref	%	time/ref	%	time/ref
0	3.0	46	.14	31	.10	6.0	.02	10	.032
1	7.7	48	.20	29	.12	4.7	.02	13	.056
2	11	49	.20	28	.11	4.7	.02	13	.053
4	26	49	.20	28	.11	4.6	.02	13	.054
8	47	49	.21	27	.12	4.3	.02	13	.057
16	89	48	.21	26	.12	4.4	.02	14	.061

Table 4: Profiling of Hashing Algorithm, Mult2 Traces

Table 4 shows the breakdown of Mult2 run time done with gprof for our hashing algorithm. % indicates the total amount of time spent in that activity as a percentage of the total time to process entire trace. Time/ref is in virtual msec.

all cases, our simulator required less than 400K bytes (a maximum stack of 2049 elements) for all data structures, of which about 10% went to the hash table.

Summing up, hashing algorithms can do INPUT, FIND, METRIC, and UPDATE in essentially constant time, and thus process a trace of length N references in essentially $O(N)$ time. However, due to constant factors from maintaining a doubly-linked list, discard-pointers, in-memory fields, and pruning and maintaining the hash table, the hashing algorithms performs better that the linked-list implementation only when the mean stack distance is above a certain threshold. For our traces, the crossover occurs for stack distances between three and five. For small mean stack distances, it is probably most appropriate to use linked-list based simulators. This is especially true for CPU cache simulations, where small associativities limit stack depths. But, for traces with poor locality, like disk or file system traces, hashing is the better choice. Thompson suggests that file and disk system traces have mean stack distances between 200 and 500 [Tho87]. Our results indicate that hashing would give ten to twenty times the performance of a linked-list-based simulator for these traces.

4. Extensions and Limitations

The hashing algorithm presented in this paper can be extended to work with memories that use write back of dirty blocks to reduce memory traffic. Thompson and Smith [ThS89] have proposed an efficient method for recording the number of write backs, by recording a *dirty level* within each block. The dirty

level indicates the stack distance at or below which all memories have that block dirty. We can then maintain for each memory size the number of *writes avoided*, or writes to a block which is already dirty. We increment the writes avoided for a particular memory size whenever a write occurs to a block for which this is the smallest memory equal to or larger than the block's dirty level. Since a write to a block that is already dirty in a memory will not cause another write back, we can calculate the true number of write backs that must take place for a given memory size of interest by subtracting the sum of the writes avoided for this and smaller sizes from the total number of writes that occurred in the course of the simulation (also, the number of dirty blocks which remain in the memory at the end of the simulation must be subtracted if we assume that the memories are not flushed at the end of the simulation). A key point in incorporating write backs into our algorithm is that we need to update a block's dirty level only when it is referenced.

This hashing algorithm is not adaptable to all-associativity simulation [HiS89]. All-associativity simulation relies on traversing the stack and recording the number of blocks that would have appeared in the same set as the referenced block over a range of number of sets. Since the hashing algorithm does not traverse the stack, it does not generalize to all-associativity simulation without losing the improvements in simulation speed for which hashing was introduced.

We assumed an LRU replacement policy from the outset. Hashing as presented in this paper is not adaptable to other stack algorithms besides LRU without suffering degradation in simulation speed. Although hashing can be applied in general to allow blocks to be found in a stack in constant time, our hashing algorithm relies on the orderly fashion in which LRU moves blocks between memory sizes to allow efficient maintenance of the in-memory fields.

5. Conclusion

In this paper we have described a new algorithm for efficiently simulating highly-associative memories, such as disk and file-system caches. The algorithm also performs efficiently for CPU address traces reduced using stack deletion. These traces all exhibit large mean stack distances, resulting in long simulation times using a conventional stack algorithm. Conventional stack algorithms have run-times of $O(N \times D)$, where N is the trace length, and D is the mean stack distance. Several other stack algorithms use tree structures to reduce the running time, but achieve bounds of at best $O(N \log D)$. Our algorithm uses hashing to find references in the stack and their stack distances (the FIND operation) in constant time, and hence requires essentially constant time per reference or $O(N)$ for the entire trace.

The key to the algorithm is that computer designers are not interested in a continuum of memory sizes, but rather in discrete sets of sizes (e.g., the powers of two). The algorithm exploits this fact, and uses

in-memory fields and discard-pointers to determine the smallest memory currently containing the requested block (i.e. its stack distance). Thus the algorithm runs in $O(N\sum_{i=1}^{i=M} m_i)$ time, where the m_1 through m_M are the miss ratios for the M memory sizes under investigation. In most cases, researchers investigate a small number of memory sizes (e.g., $M \leq 8$) and $\sum_{i=1}^{i=M} m_i$ is much less than M, since highly associative memories tend to have low miss ratios. In any case, since $\sum_{i=1}^{i=M} m_i$ cannot exceed M, a constant with respect to trace locality, our algorithm's run-time is simply $O(N)$. Furthermore, our algorithm is a simple extension to the traditional linked-list algorithm, and hence is much easier to code than the tree-based schemes.

In this paper, we compared an implementation of our hashing stack algorithm to a link-list-based simulator. Run-time measurements showed that the traditional algorithm works best for small mean stack distances, because of the additional hash table overhead. But the new algorithm quickly dominates as the mean stack distance increases above approximately 5. Thus our algorithm is clearly superior for disk and file-system traces, which have mean stack distances in the hundreds.

In the future, we plan to extend this work by comparing the hashing and link-list algorithms for disk and file-system traces. We also hope to measure the run-time performance of at least one tree-based algorithm to allow a more complete comparison.

6. Acknowledgements

We would like to thank Anita Borg, David Wall, and DEC WRL for use of traces collected there, and Richard Kessler for work done in collecting and de/compressing the traces used and for reading and improving drafts of this paper. We would also like to thank Madhusudhan Talluri and Tony Laundrie with help in running simulations.

7. 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.

[HiS89] M. D. Hill and A. J. Smith, Evaluating Associativity in CPU Caches, *IEEE Transactions on Computers 38*, 12 (December 1989), 1612-1630.

[HoS76] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Inc., 1976.

[Knu73] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1973.

[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.

[ODH85] J. Ousterhout, H. DaCosta, D. Harrison, J. Kunze, M. Kupfer and J. Thompson, A Trace-Driven Analysis of the UNIX 4.2 BSD File System, *Proceedings Tenth Symposium on Operating System Principles*, Orcas Island, Washington, December 1985.

[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.

[ThS89] J. G. Thompson and A. J. Smith, Efficient (Stack) Algorithms for Write-Back and Sector Memories, *ACM Trans. on Computer Systems 7*, 1 (February 1989), 78-116.