STACK SIMULATION

Trace-driven simulation is a standard process for analyzing memory system performance. Given a sequence of memory addresses, trace-driven simulation is often used to produce performance metrics such as miss and/or write-back ratios. These results can then be incorporated into a system model to predict sytem performance under different memory design choices. The time-consuming trace-driven simulation is usually repeated for many alternative memory configurations with multiple traces obtained from various environments, making the whole process very tedious and more time consuming.

Stack simulation is a one-pass simulation technique for analyzing memory system performance. It is a powerful approach to generate miss ratios and write-back ratios for multiple memory configurations in a single run. In this article, we first introduce the general stack simulation algorithm. Since cache is an important part of any modern memory system, we will use cache as an example of the memory system, and then devise efficient stack simulation algorithms to speed up trace-driven cache simulation.

Mattson and his colleagues (1) proposed an efficient trace-driven simulation method for storage hierarchies. Their approach, called *stack algorithm,* uses a common stack to store references and allows miss ratios for all cache sizes to be computed in a single pass over the reference trace. Traiger and Slutz (2) extended the stack algorithm to compute miss ratios for variable line sizes and variable associativity in a single pass.

The stack algorithm works as long as the replacement policy, such as the LRU (least recently used) replacement policy, guarantees the inclusion property, that is, the contents of any size cache is a superset of those in a smaller cache. For fully associative caches, the references at any time can be represented as a stack, with the upper d elements of the stack representing the lines present in a cache with d blocks (cache lines). For a reference that is in the stack with stack distance d from the top of the stack (where the stack distance is one), all caches with d blocks or more would have a hit. Figure 1 shows the basic concept of stack simulation for fully associative memories.

The basic block in the stack could be a cache line or a main memory page. In Fig. 1 the blocks a, b, c, and d are accessed one at a time in sequence. The top of the stack always holds the most recently accessed block, regardless of memory size. In addition, the replacement policy is LRU, that is, the least recently used block is replaced by a new block.

For fully associative memories, there is only one stack and the stack simulation runs as follows: For each referenced block one searches the stack for the block until it is found or until the end of the stack is reached. During the search each block in the stack is compared with the referenced block, and a vector of right-matched counters, $M(i)$, is used to record the number of blocks in the stack with exactly i right-matched bits.

Figure 2 shows a five-block stack and current referenced block with address *1001000*. The top block address also ends with binary *00*, that is, having two right-matched bits compared to the current reference. Therefore, the right-matched counter $M(2)$ is incremented from 0 to 1. Each block encountered during the stack search increments one and exactly one

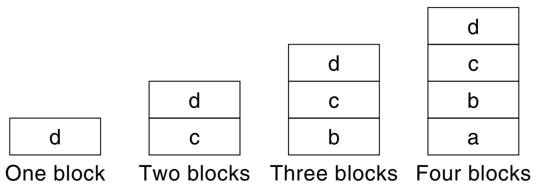

Figure 1. Stack representation of fully associative memories with up to four basic blocks and access sequence a, b, c, d.

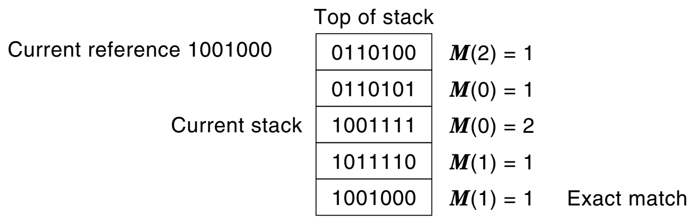

Figure 2. Increment right-matched counters $M(i)$, $i \geq 0$ and calculate stack distance $d(n)$, $U \geq n \geq 0$. Stack distance $d(3) = M(7) = 1$; $d(2) = M(2) + M(7) = 2$; $d(1) = M(1) + M(2) + M(7) = 3$; and $d(0) = M(0) + M(1) + M(2) + M(7) = 5$.

right-matched counter, until the reference is found or the end of the stack is reached. The reference in Fig. 2 was found at the bottom of the stack. If the reference is found in the stack, it is moved to the top of the stack; otherwise a new stack element is created at the top of the stack using the reference. When the referenced block is found, the right-matched counters are used to calculate the *stack distance* of the block. The right-matched counters are reset to zero at the beginning of each stack search for the next reference.

Stack distance is defined as the position of a block in a virtual stack, which represents a set of blocks, such as a cache set. If the block is found at the top of the stack, its stack distance is 1. Since multiple caches with different number of sets are usually considered during a stack simulation, stack distances of a reference may be different in different caches. In Fig. 2, $d(0) = 5$ indicates that the reference 1001000 would be found at the fifth position in a fully associative cache (or memory). Similarly, $d(1) = 3$ indicates that the reference would be found in the third position of the set in a cache with two sets. Likewise, $d(2) = 2$ and $d(3) = 1$ indicate that the reference would be found in the second position of the set in a cache with four sets, and in the first position of the set in a cache with eight sets. Figure 3 shows the position of the current referenced block if the stack is divided into two or four sets.

If a block y is ahead of x in a 2^n-set cache, it will also be found ahead of x in a 2^i-set cache, $0 \leq i < n$. Note that a cache with only one set is a fully associative cache. Since each block is associated with an address in the stack, blocks in the stack with exactly n right-matched bits would be found in the same set in a 2^i-set cache, $i \leq n$. If the current reference x is found in the stack, the distance counters $M(i)$, $0 \leq i \leq U$, have been incremented for all elements ahead of the line. Therefore, the stack distance of x in a 2^n-set cache is the number of elements found in the stack with at least n right-matched bits, that is,

$$d(n) = \sum_{i \geq n} M(i) \tag{1}$$

Thus the current reference x is a hit in a 2^n-set s-way cache if the set-associativity $s \geq d(n)$, otherwise a miss. This implies that we can calculate miss ratios for multiple cache configurations in a single simulation run.

STACK SIMULATION

The number of bits in an address may be big, which in turn requires a long vector for right-matched counters. In addition,

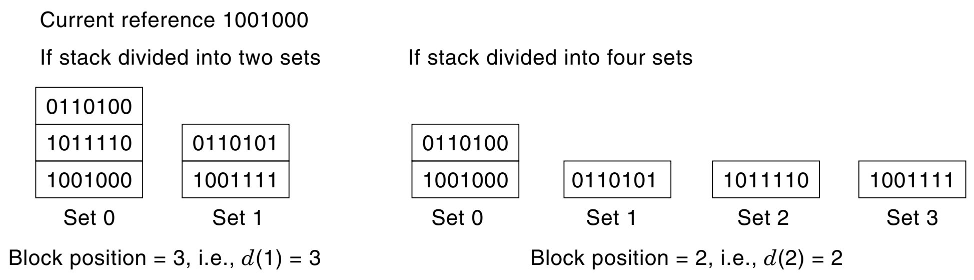
Current reference 1001000
If stack divided into two sets
0110100
1011110
1001000
0110101
1001111
Set 0
Set 1
Block position = 3, i.e., $d(1) = 3$
If stack divided into four sets
0110100
1001000
0110101
1011110
1001111
Set 0
Set 1
Set 2
Set 3
Block position = 2, i.e., $d(2) = 2$

Figure 3. Block position if the stack is divided into two or four sets.

searching a single common stack for every reference may be still time consuming. To improve the basic stock simulation technique, one needs to limit the cache configurations under investigation.

In reality, the cache configurations under investigation is likely limited. For example, the number of sets in the cache may vary from 2^L to 2^U sets. Modern processor caches typically have set-associativity not greater than 8. If a cache has 1024 cache lines, the minimal number of sets would be 128 (i.e., $L = 7$) for an eight-way cache, and the maximal number of sets would be 1024 (i.e., $U = 10$) for a direct-mapped cache.

The lower bound L, if it is not zero, provides an opportunity to speed up the simulation with multiple stacks. Multiple stacks reduce the length of each stack, thus reducing the time needed to search for a new block. For example, $L = 1$ indicates that one may use two stacks instead of one, and for each reference a stack is chosen based on the right-most binary digit of the reference address. In this case all the blocks in the stack will have at least one right-matched bit when compared to the reference address. Thus, there is no need to use right-matched counters $M(i)$, $i < L$.

The upper bound U also limits the number of right-matched counters needed during each stache search. If the number of sets under investigation is at most 2^U, there is no need to keep right-matched counters $M(i)$, $i > U$. In this case, whenever there are more than U right-matched bits between the block in the stack and the newly referenced address, one increments $M(U)$.

Let S be the maximal set-associativity under investigation. A distance counter, $D(n, i)$, where $L \le n \le U$ and $1 \le s \le S$, is used to record the number of references that have stack distance $d(n) = s$. Stack distances are calculated using Eq. (1). The distance counters are then used to calculate the number of misses for different cache configurations. Assuming the total number of references is N, the number of misses for a 2^n-set s-way cache is given by

$$\text{miss}(n, s) = N - \sum_{i=1}^{s} D(n, i) \tag{2}$$

Figure 4 shows the generic stack algorithm with multiple stacks to calculate miss ratios in a single run. At line 4 a stack is chosen for search. During the stack search the number of right-matched bits are determined, and the corresponding right-matched counter is incremented. At lines 11, 12, and 13, stack distance d is calculated starting from $d(U)$. Only one variable d is used for $d(n)$, $U \ge n \ge L$. At line 16, the algorithm calculates the number of misses for each cache configuration based on Eq. (2).

STACK SIMULATION FOR WRITE-BACK CACHES

Write-back caches are different from write-through caches. The latter modifies the next-level memory when a cache line is written, while the former does not modify the next-level memory until the dirty cache line is replaced. Write-back caches usually reduce bus traffic at the expense of additional hardware. To quantify bus traffic for write-back caches, the number of write-backs is as important as the number of misses.

There have been a number of papers in the literature dealing with efficient trace-driven cache simulation based on the stack algorithm. In particular, Thompson and Smith (3) proposed an efficient stack algorithm to obtain the number of write-backs, as well as the number of misses for fully associative caches, by attaching a dirty level to each element in the stack. A dirty level is a kind of stack distance associated with each block in a stack to indicate how far the dirty block has been pushed into the stack. Since a cache line may have different stack distances in caches with different numbers of sets, it would have different dirty levels as well in these caches. Wang and Baer (4) extended the approach by attaching a vector of dirty level rather than a single dirty level.

A vector of dirty level $L(i)$, where $L \le i \le U$, is required for each block in a stack to calculate the number of write-backs. A dirty level 0 indicates a clean line, while a nonzero dirty level indicates how far the dirty line has been pushed

```
1.   FOR each reference X DO {
2.     N++
3.     M(i) = 0 for all i, L <= i <= U
4.     a = X mod (2 ** L)
5.     search stack(a) {
         /* for each element Y in stack */
6.       b = number of right-matched bits (X vs Y)
7.       M(min(b, U))++
8.     } until X is found or end-of-stack
9.     IF found THEN {
10.      d = 0
11.      FOR (i = U; i >= L; i--) DO {
12.        d = d + M(i)
13.        DC(i, d)++
         }
14.      move the line to top of stack(a)
       } ELSE {
15.      bring new tag X to top of stack(a)
   } }
16. gather statistics
```

Figure 4. Stack algorithm.

into the stack. When a missed line is brought into a stack, the elements of its dirty level vector are initialized to zeroes if it is a read miss, or to ones if it is a write miss. The dirty levels of a clean line remain zero until the line is written; in such a case the line is brought to the top of the stack and its dirty levels are updated to ones.

For a reference X which is found in the stack with dirty level $L(n) \leq 1$ and stack distance $d(n)$, its dirty level for a 2^n-set cache is given by $L(n) = \max[d(n), L(n)]$. That is, its dirty level $L(n)$ is increased to its current stack distance $d(n)$ for a 2^n-set cache if $d(n) > L(n)$; otherwise $L(n)$ remains unchanged.

To calculate the number of write-backs for caches with different numbers of sets and set-associativity, Wang and Baer (4) use an array of write-avoided counters $A(n, s)$, where $L \leq n \leq U$ and $1 \leq s \leq S$, to record the number of writes that have $L(n) = s$. For a write access with $L(n) = s$, a 2^n-set cache having a set-associativity s or larger would have the dirty line, that is, the reference doesn't have to be written back. Therefore the write-avoided counter $A(n, s)$ is incremented by one. After the write-avoided counters have been updated for a write access, its dirty level vector is then set to all one, in that the dirty line is now at the top of the stack. Let the total number of write accesses be W. The number of write-backs for a 2^n-set s-way cache is given by

$$\text{wback}(n, s) = W - \sum_{i=1}^{s} A(n, i) \tag{3}$$

Figure 5 shows the stack algorithm to obtain miss and write-back ratios for write-back caches. Use $L(i)|Y$ to represent the dirty level $L(i)$ of a block Y in the stack. At line 15 of Fig. 5, the dirty level is updated to show how far the line has been pushed into the stack if the line is dirty. At lines 17 and 18, the write-avoided counter is incremented before moving the line back to the top of the stack, where its position is 1.

Stack simulation is a powerful approach to generate cache performance metrics in a single run. It takes a sequence of addresses, called an *address trace,* as input. Since trace files are large, and since typically a number of cache configurations are under investigation, it makes sense to use stack simulation techniques to speed up such a tedious performance analysis process.

CACHE TYPES AND ADVANCED STACK SIMULATION

Trace-driven simulation takes a sequence of addresses as input. In many cases the addresses are either virtual addresses generated from a software simulator, or real addresses collected from a piece of hardware that monitors bus addresses. While page offsets are the same in a virtual address and its corresponding real address, the virtual page number is usually different from its corresponding real page number. A real address is an address that requires no address translation, or it is referred to as a virtual address. For a small cache whose size is not larger than its set-associativity times the page size, the high-order bits in the page offset are enough to index the cache. For example, if the page size is 4 kbytes and cache line size is 32 bytes, a four-way 16 kbyte cache would have 128 sets, requiring 7 bits to choose a set out of the 128 sets. The 7 most significant bits in the page offset (12 bits in total for a 4 kbyte page) is enough to index the cache.

The result of a trace-driven simulation is only as good as its input trace. If the input trace makes no sense for the cache configuration, the result may be meaningless. For small caches, real address traces usually provide more accurate simulation results. Even if real address traces are not avail-

```
1.   FOR each reference X DO {
2.     N++
3.     IF write THEN W++
4.     M(i) = 0 for all i, L <= i <= U
5.     a = X mod (2 ** L)
6.     search stack(a) {
         /* for each element Y in stack */
7.       b = number of right-matched bits (X vs Y)
8.       M(min(b, U))++
9.     } until found or end-of-stack
10.    IF found THEN {
11.      d = 0
12.      FOR (i = U; i >= L; i--) DO {
13.        d = d + M(i)
14.        DC(i, d)++
15.        IF L(i)|Y != 0 THEN L(i)|Y = max(d, L(i)|Y)
16.        IF write THEN {
17.          A(i, L(i)|Y)++
18.          L(i)|Y = 1
         } }
19.      move the line to top of stack(a)
       } ELSE {
20.      bring new tag X to top of stack(a)
21.      IF read THEN L(i)|X = 0, for all L <= i <= U
22.      ELSE /* write */ L(i)|X = 1, for all L <= i <= U
   } }
23. gather statistics
```

Figure 5. Stack algorithm for write-back R/R-type caches.

able, virtual address traces are commonly used in trace-driven simulation to estimate cache performance and still provide reasonable estimated results.

On the other hand, modern caches may be much larger than memory page size, and so extra index bits are needed. The index bits may come from the virtual page number or real page number. In addition, the address tags stored in the cache directory may also be either virtual or real. To classify cache types, both index and tags are used. Since both the index and tag could be virtual and real, there are four possible combinations or types, namely, R/R, V/R, V/V, and R/V (5).

R/R-type caches, indexed by real addresses and having real tags, are used in many conventional computer systems, including the VAX 11/780 (6), IBM 3033 (7), and MIPS R3000 (8). Since the page offset in a virtual address is the same as that in the real address, many small R/R-type caches use bits from the page offset to select a set of cache lines while simultanously performing virtual-to-real address translation. All small caches with size not greater than its set-associativity times page size are considered as R/R-type caches. Conversely, a cache indexed by virtual addresses and having virtual tags is called a V/V-type cache, such as those in the Berkely SPUR (9), the MU-5 (10), and the first-level cache in MIPS R6000 (11).

Caches need not be completely virtual or real. The cache in IBM 3090 system (12), for example, is indexed by virtual addresses, while real address tags are used in the cache directory to compare with translated real addresses from the TLB. Such a cache is a V/R-type cache. Similar to V/V-type caches, the advantage of a virtually indexed cache is the inherent parallel accesses to the table look-aside buffer (TLB) and the cache array. Other existing commercial systems having virtually indexed caches with real tags include the HP Precision Architecture (13), MIPS R4000, and ELXSI 6400.

The fourth combination is R/V-type caches, indexed by real addresses and having virtual tags. An example of this cache type can be found in (14). However, so far there are no known commercial computer systems having an R/V-type cache.

A comprehensive study on various cache types can be found in (5). In addition to the four basic cache types, there have been cache designs merging two basic cache types, in which one path is usually faster than the other, to provide a short cut on a cache hit and/or provide additional support for cache coherence. For example, Goodman (15) proposed a cache with two directories. One cache directory is indexed by real addresses and uses real tags, and the other indexed by virtual addresses and uses virtual tags, that is, a combination of an R/R-type cache and a V/V-type cache. The second-level cache is MIPS R6000 (14) is basically an R/R-type cache, which also has virtual tags in its cache directory for fast accesses on cache hits. Thus it is a combination of an R/R-type cache and an R/V-type cache. Baylor and colleagues (16) proposed another cache whose directory contains both real and virtual tags, but indexed by virtual addresses rather than real addresses—a combination of a V/V-type cache and a V/R-type cache.

If address traces contain both virtual addresses and their corresponding real addresses, such as the trace files used in (17), which were collected in an IBM mainframe, more accurate simulation results may be obtained for non-R/R-type caches. In particular, more than one virtual address may be mapped (or translated) to the same real address in modern computer systems, causing problems known as *synonyms* in virtually indexed caches. When two virtual addresses map to the same real address and the line has been brought into a virtually indexed cache using one of the virtual addresses, the set selected by the current virtual address is called the *primary set,* while all other sets in which the synonym line might be found are called the *synonym sets.*

In a V/R-type cache, synonym may or may not cause any problems. In fact, it is a primary hit as long as the line is found in the primary set. If the synonym line is not found in the primary set, additional cycles may be needed to access synonym sets for the cache line. In order to distinguish different cases in a V/R-type cache, *alias* is defined as the case in which the two virtual addresses differ only in the bit positions that are not used to index the cache, and *pseudonym* is defined as the case in which the two virtual addresses differ in the bit positions used to index the cache. In other words, it is an *alias* when the line is found in the primary set, and a *pseudonym* when the line is found in a synonym set.

Given traces with both virtual and real addresses, advanced stack simulation algorithm (17) has been developed to calculate the number of pseudonym as well as misses and write-backs for V/R-type caches in a single run. The algorithm may be extended further for R/V-type and V/V-type caches, or a combined cache with two basic types.

CONCLUSION

Stack simulation is a powerful approach to calculate performance metrics for a number of cache configurations in a single run. It saves the total simulation time and makes trace-driven simulation less tedious.

BIBLIOGRAPHY

1. R. Mattson et al., Evaluation techniques for storage hierarchies, *IBM Syst. J.,* **9** (2): 78–117, 1970.
2. I. Traiger and D. Slutz, One pass technique for the evaluation of memory hierarchies, *IBM Res. Rep.,* RJ 892 (#15563), July 1971.
3. J. Thompson and A. Smith, Efficient (stack) algorithms for analysis of write-back and sector memories, *ACM Trans. Comput. Syst.,* **7** (1): 78–117, 1989.
4. W.-H. Wang and J.-L. Baer, Efficient trace-driven simulation methods for cache performance analysis, *ACM Trans. Comput. Syst.,* **9** (3): 222–241, 1991.
5. C. E. Wu, Y. Hsu, and Y.-H. Liu, A quantitative evaluation of cache types for high-performance computer systems, *IEEE Trans. Comput.,* **42**: 1154–1162, 1993.
6. D. Clark, Cache performance in the VAX 11/780, *ACM Trans. Comput. Syst.,* 24–37, 1983.
7. IBM, *IBM 3033 Processor Complex: Theory of Operation/Diagrams Manual,* vol. 4, Poughkeepsie, NY: IBM, 1978.
8. T. Riordan et al., System design using the MIPS R3000/3010 RISC chipset, *Dig. Papers, Spring 1989 IEEE Compcon,* 1989, pp. 494–498.
9. M. Hill, Design decisions in SPUR, *IEEE Comp.,* **19** (11): 8–22, 1986.
10. R. Ibbet and M. Husband, The MU5 name store, *Comput. J.,* **20** (3): 227–231, 1977.
11. D. Roberts, T. Layman, and G. Taylor, An ECL RISC microprocessor designed for two level cache, *Proc. Int. Conf. Comput. Design,* 1990, pp. 228–231.

12. S. G. Tucker, The IBM 3090 system: An overview, *IBM Syst. J.,* **25** (1): 4–19, 1986.
13. R. Lee, Precision architecture, *IEEE Comput.,* **22** (1): 78–91, 1989.
14. G. Taylor, P. Davies, and M. Farmwald, The TLB slice—A low-cost high-speed address translation mechanism, *Proc. 17th Int. Symp. Comput. Architecture,* 1990, pp. 355–363.
15. J. Goodman, Coherency for multiprocessor virtual address caches, *Proc. 1987 ASPLOS,* 1987, pp. 72–81.
16. S. Baylor, K. McAuliffe, and B. Rathi, An evaluation of cache coherence protocols for MIN-based multiprocessors, *Proc. Int. Symp. Shared Memory Multiprocess.,* 1991, pp. 1–32.
17. C. E. Wu, Y. Hsu, and Y.-H. Liu, Stack simulation for set-associative V/R-type caches, *Proc. 16th Comput. Softw. Appl. Conf.,* 1992, pp. 1154–1162.

Reading List

M. Hill and A. Smith, Evaluating associativity in CPU caches, *IEEE Trans. Comput.,* **C-38**: 1612–1630, 1989.

C. Eric Wu
Yarsun Hsu
Yew-huey Liu
IBM T. J. Watson Research Center

STANDARD CELLS. See Voltage references.
STANDARDS. See Handbooks and standards.
STANDARDS, DOCUMENT INTERCHANGE. See Document interchange standards.