RC 21038
Computer Sciences/Mathematics

IBM Research Report

Techniques for Fast Simulation of Associative Cache Directories

Mayan Moudgill
mayan@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden ● Austin ● China ● Haifa ● Tokyo ● T.J. Watson ● Zurich

Techniques for Fast Simulation of Associative Cache Directories

Mayan Moudgill
mayan@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

We describe a technique for simulating associative cache directories that considerably reduces simulation time with respect to a sequential search of the tag array. We also describe techniques for maintaining LRU information that use considerably less memory than time-stamps. The combination of these techniques makes possible the simulation of set-associative cache directories at a much higher speed than other techniques.

1. Introduction

When modeling processors, the time spent in simulating the cache directories can be a considerable portion of the total simulation time. This is specially true when attempting to model processors with multiple levels of caches and translation look-aside buffers (TLBs); for instance, the model for a processor with independent first level I & D caches, a second level unified cache and separate I & D TLBs, contains 5 caches. Consequently, speeding up the simulation of cache directories will speed-up the simulation of the overall model.

The literature on enhancing the simulation performance of a single cache model is fairly sparse. Most of the published work in this field concentrates on speeding up the simultaneous simulation of multiple caches [1]. This may have been due to the fact the overhead of producing and reading address traces and simulating other portions of the model has dominated the simulation time.

As part of our work on microprocessor organizations, we have developed tools that are so efficient that the cache directory simulation time becomes a significant portion of the overall time. In particular, Turandot [2], our model of complex out-of-order processor, simulates over 100 million processor cycles per hour on a standard workstation. We found that, with these tools, we could no longer ignore the impact of cache directory simulation.

One source of performance improvement is to decrease the time spent searching to find a hit or miss in a cache directory. The only approach we encountered in the literature that improves upon the performance of a sequential search of the tag array is described in [4]; it takes advantage of the properties of the most-recently used (MRU) entries of sets to decrease the average number of instructions executed per cache lookup. We have developed a lookup algorithm, the *preferred location* algorithm, that potentially executes fewer instructions on a cache lookup than either a sequential search or the MRU based scheme.

As is widely known, the overall performance of any program is dependent on the frequency of its cache misses. This is, in general, proportional to the amount of memory used by the program. Traditionally, cache directory simulations have used an extra word per entry to hold a time-stamp - i.e. a counter indicating the cycle on which the entry was last accessed. This is used to compute the least-recently used (LRU) entry during cache line replacement. We have developed techniques for encoding LRU information for 2, 4 and 8-way set-associative cache directories that use less memory than a time-stamp technique. This should reduce the number of cache misses during simulation, and thereby further improve simulation performance.

In this paper we shall:

- describe the basic algorithms for simulating set-associative caches - sequential search and MRU;
- describe our new algorithm, and compare it with the existing ones; and
- describe techniques for computing LRU for 2, 4 and 8-way set-associative caches that use less memory than time-stamps.

2. Existing Algorithms

2.1 Direct Mapped Cache

First, let us consider the simulation of the directory for a direct mapped cache. The function we will consider, **`probe_cache()`**, checks to see if an address hits in the cache or not. If the address hits in the cache, it returns 1, otherwise, it returns 0. Furthermore, in the case of a miss, the directory state is altered to reflect the loading of the line containing the missed address.

The cache being simulated is assumed to compute a tag and a set selector as shown in Figure 1; i.e. the tag

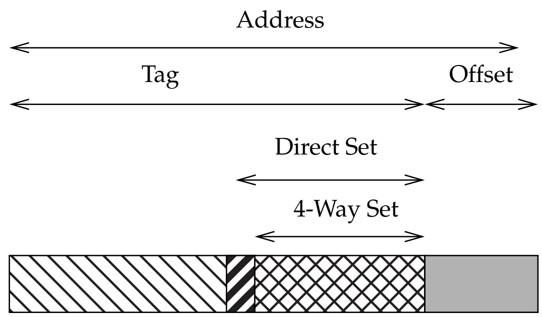

Figure 1: Cache access

consists of the most significant bits of the address, and the set is the least significant bits of the tag. In this case, the code for the function is as follows:

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned set = tag & DIRECT_SET_MASK;
  if( cache[set] == tag) {
    /* HIT! */
    return 1;
  }
  /* MISS */
  /* update the cache */
  cache[set] = tag;
  return 0;
}
```

As can be seen, the number of instructions executed by the routine when there is a hit in the cache is very low; in the case of the PowerPC architecture, a hit can be processed in 7 instructions.

2.2 4-Way Set-associative Caches

In the direct mapped cache, only one location needs to be examined to determine whether the address hits or misses in the cache. In the case of a 4-way set-associative cache, the situation is different: in the worst case, four locations need to be examined. Simulating 4-way associativity using the traditional approach of sequentially searching the four locations yields the following function:

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned set = tag & 4WAY_SET_MASK;

  if( cache[set] == tag) {
    return 1;
  }
  if( cache[set+SET_SIZE] == tag ) {
    return 1;
  }
  if( cache[set+SET_SIZE*2] == tag ) {
    return 1;
  }
  if( cache[set+SET_SIZE*3] == tag ) {
    return 1;
  }
  /* MISS */
  /* update the cache */
  lru_locn = ...
  cache[ set + SET_SIZE * lru_locn ] = tag;
  return 0;
}
```

In the PowerPC architecture, detecting a hit with this code takes 17 instructions in the worst case and about 12 instructions on the average.

An additional source of performance degradation is the branches, which are taken somewhat randomly. This can cause branch prediction hardware to mispredict, thereby further degrading simulation performance.

An improvement on the sequential search algorithm is the scheme described in [4], which we shall call the MRU algorithm. This algorithm keeps the most recently used entry of a set in the first location to be searched. Thus, if an entry is found in the third location, it is swapped with the first.

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned set = tag & 4WAY_SET_MASK;

  if( cache[set] == tag) {
    return 1;
  }
  if( cache[set+SET_SIZE] == tag ) {
  /* SWAP set with set + SET_SIZE */
    value = cache[ set ];
    cache[ set ] = cache[ set + SET_SIZE ];
    cache[ set + SET_SIZE ] = value;
   return 1;
  }
  if( cache[set+SET_SIZE*2] == tag ) {
  /* SWAP set with set + 2*SET_SIZE */
    ...
   return 1;
  }
  if( cache[set+SET_SIZE*3] == tag ) {
  /* SWAP set with set + 2*SET_SIZE */
    ...
   return 1;
  }
```

```
    /* MISS */
    /* update the cache */
    replc_locn = ...
    cache[ set + SET_SIZE * replc_locn ] =
      cache[ set ];
    cache[ set ] = tag;
    return 0;
  }
```

In the MRU algorithm, hits in the first location lead to the same amount of work as in the sequential search algorithm. In the other cases, extra work is involved in swapping the locations. It is much more likely that the next reference in a set will be the MRU address. Thus, the MRU approach trades-off the extra work required if the address is not found in the first location against the improved likelihood that it will be found on the first probe.

3. The Preferred Location Algorithm

We now describe our approach to improving the performance of simulating associative caches directories. The basic premise is this: *try to keep a tag in the same position it would have been in if the cache were direct mapped*. We shall call this position the *preferred location* for that tag.

Note that the set selector in a 4-way associative cache uses 2 fewer bits than in a direct mapped cache of the same size. When looking for a tag in the cache directory, we first index into the directory using those 2 bits as well. If there is a hit, then the probe is done and has achieved the same performance as a direct mapped cache.

In case the tag is not found, all locations for the set may need to be examined. If the tag is found in another location, it will be swapped with the tag in the preferred location. If there is a miss, the tag will be loaded into the preferred location, and the tag that was in the preferred location will be moved into the LRU location.

Note that we have chosen the four locations associated with a set as shown in Figure 2; the preferred location for any tag belonging to that set always maps to one of those four locations. The resulting algorithm can be implemented as follows:

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned pref = tag & DIRECT_SET_MASK;
  unsigned set = tag & 4WAY_SET_MASK;

  if( cache[pref] == tag ) {
    /* HIT! */
    return 1;
  }
```

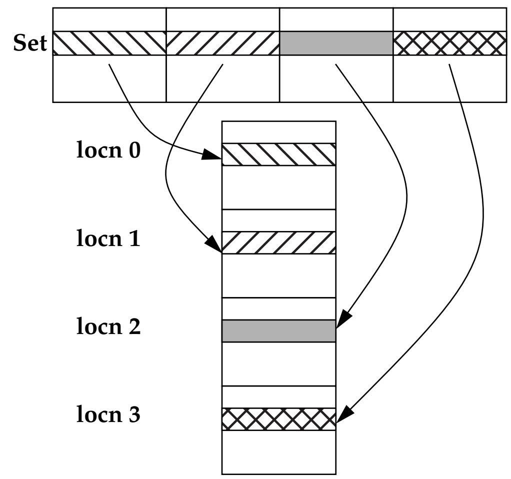

Figure 2: Set entry arrangement in preferred location

```
  if( cache[set] == tag) {
    /* swap */
    cache[set] = cache[pref];
    cache[pref] = tag;
    return 1;
  }
  if( cache[set+SET_SIZE] == tag ) {
    ... /* swap */
    return 1;
  }

  if( cache[set+SET_SIZE*2] == tag ) {
    ... /* swap */
    return 1;
  }

  if( cache[set+SET_SIZE*3] == tag ) {
    ... /* swap */
    return 1;
  }
  /* MISS */
  /* update the cache; but first swap */
  replc_locn = ...
  cache[ set + SET_SIZE * replc_locn ] =
      cache[ pref ];
  cache[ pref ] = tag;
  return 0;
}
```

In the PowerPC architecture, detecting a hit with this code requires eight instructions in those cases where the tag is found in the preferred location. In the cases where it is not, five instructions more than the straight-forward implementation are required, because of the extra initial compare and the swap. However, as will be shown later, it turns out that over 90% of the time the tag hits on the first probe.

Moreover, it turns out that when the tag misses on the first probe, it is quite often a miss. Therefore, the remaining branches in the code are strongly not taken, which can improve hardware branch predictor performance.

Benchmark	64 K 128 byte			32 K 128 byte			16 K 128 byte		
	Misses	Hits at pref	Other hits	Misses	Hits at pref	Other hits	Misses	Hits at pref	Other hits
gcc (spec 95)	0.46%	98.18%	1.35%	1.00%	96.72%	2.28%	2.29%	93.25%	4.47%
go (spec 95)	0.03%	97.92%	2.05%	0.25%	95.53%	6.23%	1.41%	89.07%	9.52%
m88ksim (spec 95)	0.09%	99.29%	0.62%	0.09%	99.18%	0.72%	0.11%	98.28%	1.61%
compress (spec 92)	11.11%	87.39%	1.50%	13.34%	85.31%	1.36%	14.84%	83.47%	1.68%
decompress (spec 92)	0.93%	94.39%	4.68%	1.96%	92.98%	5.06%	3.42%	90.14%	6.45%
OLTP	2.80%	95.37%	1.83%	4.18%	92.98%	2.84%	6.14%	89.46%	4.40%

Table 1: Preferred location behavior for 4-way set-associative data caches

While we have used 4-way associative cache directories for describing our approach, the preferred location algorithm also applies to caches with higher degrees of associativity, including fully associative caches. In fact, the advantage of the preferred location algorithm over alternative algorithms should grow with the associativity.

4. Comparisons of Simulation Performance

We measured the impact of the various cache directory simulation approaches using the following PowerPC traces:

- gcc: from the SPEC 95 suite, running the training input.
- go: from the SPEC 95 suite, running the training input.
- m88ksim: from the SPEC 95 suite, running the training input.
- compress: from the SPEC 92 suite, compressing the reference input once.
- decompress: compress from the SPEC 92 suite, decompressing the reference input once.
- OLTP: an on-line transaction processing application.

These traces were produced using `xlc` with the `-O2` optimization flag. They range in size from 44M instructions for decompress to 1,396M instructions for gcc. Only the data cache directory was simulated.

Table 1 depicts, for 4-way set-associative caches ranging in size from 16K to 64K, the percentage of probes that were misses, hits in the preferred location, and hits in some other locations. As inferred from this table, in the majority of cases, an address that hits in the cache is found at the preferred location. When the address is not at the preferred location, the address is usually not at any of the other locations.

The difference in the simulation performance of the various algorithms, of course, depends upon the implementation. A somewhat abstract metric of the relative performance is the average number of locations that need to be examined on every probe. This data is shown in Table 2. As could be expected, both MRU and preferred location algorithms examine far fewer locations per address than the sequential search - sometimes fewer than half as many. On the other hand, the advantage of preferred location over MRU is less pronounced - it ranges from 1% to 34%. with the average around 14%.

To translate the difference in number of locations examined into performance, we must take into account the amount of work done after each probe. For instance:

- The MRU and preferred location algorithms require extra book-keeping if the address does not hit in the first location searched. This extra book-keeping includes the cost of swapping the locations to the first/preferred locations.
- If the directory is using some kind of LRU scheme, then the extra book-keeping work includes swapping the LRU information.
- In the MRU algorithm, if the address hits in the first location, the LRU information may not have to be updated.

In most cases, the savings from reducing the number of searches in the MRU and preferred location algorithms outweigh any additional book-keeping required.

It is not clear which of MRU or preferred location is better when using LRU replacement. If the LRU implementation is expensive, the advantage of sometimes not having to update the LRU information may outweigh the extra searches required. This will be particularly true in low associativity/large size caches, where the number of first location hits in the MRU algorithm gets closer to the number of first location hits in the preferred location algorithm.

Benchmark	64 K 128 byte			32 K 128 byte			16 K 128 byte		
	seq	MRU	pref	seq	MRU	pref	seq	MRU	pref
gcc (spec 95)	2.53	1.13	1.03	2.51	1.22	1.06	2.53	1.35	1.12
go (spec 95)	2.43	1.20	1.02	2.38	1.38	1.08	2.54	1.62	1.21
m88ksim (spec 95)	3.18	1.03	1.02	2.66	1.05	1.01	2.77	1.17	1.02
compress (spec 92)	1.73	1.20	1.07	3.07	1.54	1.42	2.67	1.59	1.47
decompress (spec 92)	2.90	1.43	1.36	1.90	1.30	1.11	2.30	1.43	1.17
OLTP	2.54	1.23	1.11	2.56	1.31	1.17	2.59	1.45	1.25

Table 2: Locations examined per address lookup for various algorithms

5. LRU Computation in 4-Way Directories

So far we have ignored the mechanics of keeping track of the least-recently used location in each set. One obvious method is to use a time-stamp with each entry; i.e. keep a counter that is incremented on every probe of the cache. When an entry is modified (i.e., either is hit, or is loaded), the time-stamp for that entry is updated to the current time-stamp. On a miss, the LRU location in a set is computed by examining the time-stamps for the 4 locations in the set and then picking the location with the smallest time-stamp for replacement.

This approach is efficient in the number of instructions used to keep the time-stamp up-to-date. However, it has one major drawback - the amount of memory required. This scheme ends up using a word for each entry in the directory. While simulating a 2MB cache with 128 byte lines, this means that 64K bytes of memory are needed to hold the time-stamps. Interestingly, for a cache that large, only 32K bytes are needed to hold the tags*. This means that while running the simulation on a processor such as the PowerPC 604e, which has a 64K data cache, a significant number of data-cache misses could occur during the simulation simply because the time-stamp and tag arrays do not fit in the data-cache.

We now describe an approach that uses only one byte per set; thus, for the 2MB cache discussed above, (slightly over) 4KB of memory are needed to maintain the LRU information.

If we arrange the 4 locations for a set in least-recently used order, there are 4!=24 possible orderings. We encode each order in a byte as shown in Figure 3. Thus, if the order is `<3, 0, 1, 2>` with the most recently used location being `3` and the least recently used location being `2`, then the encoding is `0_1_01_10_00b`. If location `1` is used, the ordering becomes `<1, 3, 0, 2>`, whose encoding is `0_0_00_10_00b`.

When one of the four locations is used, the LRU encoding is transformed from one of the 24 encoding to

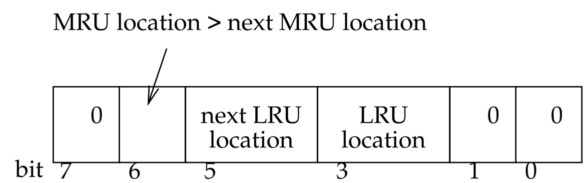

Figure 3: LRU encoding

some other encoding. This function is fairly expensive to evaluate. If the program actually evaluated this function every time a location was used, it would severely degrade performance.

Instead, the mapping for all 24x4 possible < `old_lru` | `location` > tuples is pre-computed and saved in a 128 byte table. When a location is used, resulting in a new LRU encoding, this encoding is obtained by examining the table entry at the index obtained by concatenating the old LRU with the location being used. Thus, the code in the case of a hit in location 2 is:

```
if( cache[ set + 2 * SET_SIZE ] == tag ) {
   old_lru = lru[ set ];
   new_lru = precomp_use_lru[ old_lru | 2 ];
   lru[ set ] = new_lru;
   return 1;
}
```

Note that having 0s in the 2 least significant bits of the encoding allows the index into the precomputed table to be obtained by simply OR-ing the location, in this case 2, to the old LRU value.

The encoding we are using makes the LRU location easy to find; just use bits 2 and 3 of the encoding. However, it requires that we use a 32x4 table to hold the mapping, instead of the minimum size 24x4 table.

6. 4-Way Encoded LRU & Preferred Location

When using the LRU encoding with the preferred location scheme, another table is required to deal with

* The tags are 13 bits long, and therefore can fit in a half-word.

the swapping of locations. This table, of size 4x24x4*, is indexed by the tuple **`< preferred_locn | old_lru | actual_locn >`**, and contains the following mapping:

- in the case of a hit in a location other than the preferred location, swap the positions of the hit location and the preferred location in the LRU ordering.
- in the case of a miss, swap the LRU location and the preferred location in the LRU ordering.
- after that, in both cases, move the preferred location to the MRU position in the ordering.

Putting the two schemes together yields the following code:

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned pref = tag & DIRECT_SET_MASK;
  unsigned pref_locn = pref >> SET_BITS;
  unsigned set = tag & 4WAY_SET_MASK;

  old_lru = lru[ set ];

  if( cache[pref] == tag ) {
    new_lru = precomp_use_lru[ old_lru +
      pref_locn ];
    lru[ set ] = new_lru;
    return 1;
  }
  if( cache[set] == tag) {
    new_lru = precomp_swap_lru[
      (pref_locn << 7) | old_lru | 0 ];
    lru[ set ] = new_lru;
    cache[set] = cache[pref];
    cache[pref] = tag;
    return 1;
  }
  if( cache[set+SET_SIZE] == tag ) {
    new_lru = precomp_swap_lru[
      (pref_locn << 7) | old_lru | 1 ];
    lru[ set ] = new_lru;
    return 1;
  }
  if( cache[set+SET_SIZE*2] == tag ) {
    new_lru = precomp_swap_lru[
      (pref_locn << 7) | old_lru | 2 ];
    lru[ set ] = new_lru;
    return 1;
  }
  if( cache[set+SET_SIZE*3] == tag ) {
    new_lru = precomp_swap_lru[
      (pref_locn << 7) | old_lru | 3 ];
    lru[ set ] = new_lru;
    return 1;
  }
  lru_locn = ( old_lru >> 2) & 0x3;
  new_lru = precomp_swap_lru[
       (pref_locn << 7) | old_lru |
         lru_locn ];
  lru[ set ] = new_lru;

  cache[ set + SET_SIZE * lru_locn ] =
       cache[ pref ];
  cache[ pref ] = tag;
  return 0;
}
```

There is an alternative scheme that uses only 128 bytes of extra space to handle swapping. Instead of precomputing both the swap and the use in one 512 byte table, the 128 byte table precomputes the result of swapping a location with the MRU location. In this scheme, first the old location (i.e. the hit location or the LRU location) is moved to the MRU position using the **`precomp_use_lru`** table. Then, the MRU position is swapped with the preferred location using the second table. This approach uses a quarter of the space, but requires two table lookups instead of one.

7. LRU Computation in 8-Way Directories

The 4-way LRU encoding described in the previous section does not extend to 8-way set-associative caches. Instead, we use an alternative encoding based on the hardware implementation described in [3]. Abstractly, the LRU information is encoded in an 8x8 bit matrix, where:

```
bit[ i ] [ j ] = 1, iff entry i MRU entry j
```

With this encoding, entry **`i`** is the LRU entry if **`bit[ i ] [ j ]`** is **`0`** for all **`j`**.

The 64 bit matrix is implemented as 8 bytes, one for each row, overlaid with a single 64 bit word†.

```
union {
  unsigned long long   as_word;
  unsigned char        as_byte[8];
};
```

With this representation, in the case of a hit, the LRU is updated by setting a row to all 1s and then zeroing out a column. For instance, if a tag is found at entry 2, the LRU is updated as follows:

```
if( cache[ set + 2 * SET_SIZE ] == tag ) {
  lru[ set ].as_byte[2] |= 0xff;
  lru[ set ].as_word &= ~0x0404040404040404;
  return 1;
}
```

† For simplicity, we are assuming that the simulation runs on a 64 bit processor. The same approach will work on a machine with 32 bit words, at the cost of extra instructions.

* actually 4x**32**x4

The real reason for representing the LRU information as an overlay is to make the swapping of entries easier. With the overlay representation, the code to swap the positions of entries i and j (where i > j) is as follows:

```
/* swap the rows */
byte_i = lru[ set ].as_byte[ i ];
byte_j = lru[ set ].as_byte[ j ];
lru[ set ].as_byte[ i ] = byte_j;
lru[ set ].as_byte[ j ] = byte_i;

/* swap the columns */
word = lru[ set ].as_word;
/* the bits for the i, j columns
 *   col_mask[i] is 8 bytes each with value
 *   (1<<i)
 */
bits_i = word & col_mask[ i ];
bits_j = word & col_mask[ j ];

/* move the bits to the right position in
 * the columns
 */
bits_i >>= i - j;
bits_j <<= i - j;

/* insert the bits */
word &= ~col_mask[ i ];
word |= bits_j;
word &= ~col_mask[ j ];
word |= bits_i;
```

The following code shows how to implement an 8-way set-associative cache combining the 64-bit LRU encoding with the preferred location algorithm. Note that with the preferred location algorithm, when two locations are swapped, the LRU for one of them is updated as well. This presents an opportunity to save some instructions by combining the two function. The resultant code is:

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned pref = tag & DIRECT_SET_MASK;
  unsigned pref_locn = pref >> SET_BITS;
  unsigned set = tag & 8WAY_SET_MASK;

  if( cache[pref] == tag ) {
    lru[ set ].as_byte[pref] |= 0xff;
    lru[ set ].as_word &= ~row_mask[ pref ];
   return 1;
  }
  ...
  if( cache[set+SET_SIZE*2] == tag ) {
    byte_i = lru[ set ].as_byte[ pref ];
    lru[ set ].as_byte[ 2 ] = byte_i;
    lru[ set ].as_byte[ pref ] = 0xff;

    word = lru[ set ].as_word;
    bits_i = word & col_mask[ pref ];
    if( pref > 2 ) {
      bits_i >>= pref - 2;
    }
    else {
      bits_i <<= 2 - pref;
    }

    word &= ~col_mask[ pref ];
    word &= ~col_mask[ 2 ];
    word |= bits_i;
    return 1;
  }
  ...
  /* MISS */
  for( i = 0; i < 8; i ++ ) {
    if( lru[ set ].as_byte[ i ] == 0 ) {
      lru_locn = i;
      break;
    }
  }

  byte_i = lru[ set ].as_byte[ pref ];
  lru[ set ].as_byte[ lru_locn ] = byte_i;
  lru[ set ].as_byte[ pref ] = 0xff;

  word = lru[ set ].as_word;
  bits_i = word & col_mask[ pref ];

  if( pref > lru_locn ) {
    bits_i >>= pref - lru_locn;
  }
  else {
    bits_i <<= lru_locn - pref;
  }

  word &= ~col_mask[ pref ];
  word &= ~col_mask[ lru_locn ];
  word |= bits_i;
  return 0;
}
```

The LRU encoding shown above uses 8 bytes per set, which is a quarter of the memory used by a time-stamp based approach.

It is possible to use just 4 bytes per set by exploiting the fact that the 64 bit matrix is anti-symmetric; i.e.

```
bit[ i ] [ j ] = !bit[ j ] [ i ]
```

One can take advantage of this property to encode just the lower (or upper) triangular portion of the full matrix using 28 bits. The implementation of finding the LRU and updating the encoding becomes a little more cumbersome - it requires the use of both row and column masks to extract the bits pertaining to a particular entry, but is not more inefficient. However, there is no inexpensive way of implementing the swap operation. Consequently, this encoding can only be used with the sequential tag search algorithm.

Benchmark	MRU Locn 0 Hits	Ratio wrt Pref Locn
gcc (spec 95)	89.56%	0.96
go (spec 95)	80.53%	0.91
m88ksim (spec 95)	97.39%	0.99
compress (spec 92)	79.85%	0.96
decompress (spec 92)	85.81%	0.95
OLTP	86.42%	0.97

Table 3: MRU in 2-way 16K, 128 byte line cache

8. LRU Computation in 2-Way Directories

In the case of two-way set-associative caches with LRU, the preferred location algorithm has one big drawback: the LRU information has to be maintained in a separate table, and this table has to be updated on every probe. It is somewhat more efficient to simply keep the LRU member of a set in location 0; i.e. to use the MRU algorithm. This has the additional advantage that the position of the entry encodes the LRU information - location 1 always contains the LRU entry. The following code is an implementation of 2-way set-associative caches based on this approach:

```
int probe_cache( unsigned address )
{
  unsigned tag = address >> OFFSET_BITS;
  unsigned set = tag & 2WAY_SET_MASK;

  if( cache[ set ] == tag ) {
    return 1;
  }

  if( cache[ set + SET_SIZE ] == tag ) {
    cache[ set + SET_SIZE ] = cache[ set ];
    cache[ set ] = tag;
    return 1;
  }

  cache[ set + SET_SIZE ] = cache[ set ];
  cache[ set ] = tag;
  return 0;
}
```

With this algorithm, hits in location 0 have exactly the same performance as a hit in a direct-mapped cache. As can be from Table 3, most of the time the probe does hit in location 0. Also, the preferred location algorithm does not do too much better than the MRU approach - on the average, preferred location generates only 5% more first location hits than MRU.

9. Conclusions

We have introduced the preferred location algorithm, a new way of implementing the simulation of set-associative cache directories, and compared it against two existing algorithms - sequential search, and MRU. The preferred location algorithm yields higher performance than the sequential search algorithm. In certain situations, the preferred location algorithm can also yield performance better than the MRU algorithm.

We have also shown ways of keeping track of LRU information in 2, 4 and 8-way associative caches that use considerably less memory than a more traditional timestamp based scheme.

These techniques make possible the simulation of N-way set-associative cache directories with almost the same cost as simulating a direct mapped cache directory. In particular, it appears that there is a marginal degradation in the simulation performance of 2 or 4-way set-associative cache directories with respect to direct mapped caches

10. References

[1] T.M. Conte, C.E. Gimarc, Eds, "Fast Simulation of Computer Architectures," Kluwer Academic Publishers, 1995.

[2] M. Moudgill, J. Moreno, "Turandot: a wide-issue superscalar processor model for microarchitecture exploration," Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1997.

[3] Maruyama, K, "mLRU Page Replacement Algorithm in Terms of the Reference Matrix," IBM Technical Disclosure Bulletin, March 1975, pp. 3101-3103.

[4] K. So, R. Rechtschaffen, "Cache operations by MRU change," Research Report RC 11613 REV, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1986.