Iterative Cache Simulation of Embedded CPUs with Trace Stripping

Zhao Wu and Wayne Wolf
Dept. of Electrical Engineering, Princeton University
Princeton, NJ 08544, U.S.A.
Tel: (609) 258-4261, Fax: (609) 258-3745
Email: {zhaowu, wolf}@ee.princeton.edu

Abstract

Trace-driven cache simulation is a time-consuming yet valuable procedure for evaluating the performance of embedded memory systems. In this paper we present a novel technique, called as iterative cache simulation, to produce a variety of performance metrics for several different cache configurations. Compared with previous work in this field, our approach has following features. First, it supports a wide range of performance metrics, including miss ratio, write-back counts, bus traffic, et al. Second, unlike estimation-based methods, the results produced by our simulator are accurate. Third, our approach is flexible. It can simulate both uniprocessor and multiprocessor caches, with options of higher level caches, sub-block replacement and prefetching. Last, it is fast. Our simulation results show that it has similar runtime as the fastest one-pass cache simulator.

1. Introduction

Caches are very important to embedded computer systems, especially as the gap between microprocessors and memories is continuously becoming wider and wider. Trace-driven cache simulation is a popular and essential tool for performance evaluation of memory systems. With the data-memory trace generated by an instrumented program, researchers can obtain a number of performance metrics such as cache miss ratio, write-back counts (for write-back cache), et al., and then identify the bottleneck of interested systems. This is particularly useful for designing embedded systems, where the memory plays an important part in the overall performance and must be tweaked to reduce cost. A good reference on hardware/software co-synthesis with memory hierarchies can be found in Li and Wolf's comprehensive study [1].

There are primarily two types of cache simulation. While post simulation generates a trace first and then analyzes it, on-the-fly simulation combines the two stages into a single step. There are pros and cons with both types. On one hand, post simulation requires large storage for the trace, which is not necessary for on-the-fly simulation. On the other hand, generating traces takes time; so if we want to simulate several cache architectures with the same trace collected, it is better to use post simulation.

As we know, the three basic parameters of a cache are: cache size, block size (also known as line size), and degree of associativity. In cache designs, the problem often resides in the choice of the three parameters. To get optimal performance for a program, we need to run cache simulation over the reference trace many times with varying parameters.

While the idea of trace-driven simulation is simple, it is very time-consuming since the simulation time is proportional to the length of the trace. For real applications, the traces can easily exceed millions of references. Therefore numerous efforts have been made to reduce trace length and simulation time. Among many techniques a very attractive class is one-pass simulation [2], which simulates several cache configurations in a single run. However, although the large number of methods can vastly accelerate simulation time, each of them has certain limitations. Traditional trace reduction cannot handle varying block sizes (so are most one-pass algorithms). Few one-pass algorithms support multiprocessor caches that use invalidation based coherence protocols.

In hardware/software co-design, fast simulation of several candidate cache configurations is of special importance. Li and Wolf pointed out this problem; but hampered by the time complexity of cache simulations, they only described an estimation scheme for miss rates of direct-mapped caches [1].

In this paper, we propose **iterative cache simulation**, which is a fast simulation method that can be used to evaluate a variety of cache configurations. The basic idea of iterative simulation is to arrange the caches in such a way that every time after we simulate one configuration, we can strip off some redundant information from the trace, hence to speedup following simulations. Unlike some other fast simulation algorithms which trade accuracy for time, ours produces precise results and it supports many performance metrics, including miss ratio, write-back counts, distribution of misses, et al. Even though we have to run cache simulation multiple times (each for a different configuration), results show that the total simulation time is very similar to **Cheetah**, the fastest one-pass simulator of which we are aware.

The rest of this paper is organized as follows. In section 2 we briefly review some previous work in cache simulation. Section 3 describes our method in detail. Simulation results of some multimedia applications based on our method are shown in section 4, together with the runtime of cheetah. Finally concluding remarks are drawn in section 5.

2. Related work

2.1 Trace reduction

Since the simulation time and storage requirement are directly related to the size of trace, a lot of work has been done in trying to reduce the trace length. The key idea is to retain only the references that contribute to the performance metrics. E.g., consecutive accesses to the same cache block can be shrunk to one without changing miss ratio.

Smith proposed two schemes for reducing trace length [3]. The first one, called as **stack deletion**, maintains an LRU (Least Recently Used) stack and discards references to the *D* most recently used blocks. The resulting trace can be used to obtain the miss ratio of a fully associative memory with more than *D* blocks and same block size, assuming LRU replacement is also used. The second method, called as **snapshot**, samples references at regular time intervals. These two methods were primarily used for studying paging system instead of cache.

Later, Puzak extended Smith's work to set-associative caches [4]. He proposed a technique called **trace stripping**, which passes the original trace through a direct-mapped cache and records only those accesses that cause a miss. The subsequent trace can then be used for any set-associative caches with the same block size and number of sets (number of sets = cache size / block size / associativity). Since the resulting trace only contains miss references, the workload is greatly reduced.

Combining Smith's **snapshot** method with Puzak's **trace stripping**, Agarwal and Huffman proposed **two-step filter** [5], which compacts traces through a cache filter (stripping temporal locality) followed by a block filter (stripping spatial locality). While good compression ratio is achieved with this approach, accuracy is sacrificed. Laha et al. noticed remarkable errors in large caches with small miss rates in their sampling techniques [6].

We are interested in a trace stripping technique that can always guarantee accurate performance metrics while compressing traces substantially. As pointed out by Wang and Baer, Puzak's method, albeit precise for set-associative caches in terms of miss ratio, is not sufficient for other metrics (e.g. write-back counts) or multiprocessor caches. Therefore we need to devise some other methods.

2.2 One-pass cache simulation

One-pass simulation algorithms attack the problem from another perspective. Instead of simulating one cache configuration at a time, these methods simulate a class of caches efficiently in a single pass through the trace.

As early as in 1970, Mattson et al. presented an algorithm for simulating fully associative caches with varying sizes but fixed block size [2]. Their algorithm, referred to as **stack simulation**, takes advantage of **inclusion property** (i.e. at any time of simulation, the contents of a cache is always the subset of any larger ones), which is guaranteed by certain replacement policies including LRU.

Originally stack simulation only works for fully associative caches and it only reports miss ratio. Hill and Smith identified **set-refinement property** (i.e. blocks mapped to the same set in larger caches are also mapped to the same set in smaller caches) and extended inclusion property for direct-mapped and set-associative caches. They further proposed **forest simulation** and **all-associativity simulation**, which respectively work for direct-mapped caches and caches with arbitrary associativity [7]. Meanwhile Thompson and Smith introduced **dirty level** and **writes avoided** [8], which track the status of a written block in write-back caches and give write-back counts in addition to miss ratio. Their algorithm was then generalized by Wang and Baer to set-associative caches as well [9]. The new algorithm can be used to simulate multiprocessor caches with update-based protocols, but it has some difficulties in dealing with invalidation-based protocols. Later Wu and Muntz fixed the problems by using **covering vector** and **marker splitting** to track invalidated data blocks in the stack [10].

Aside from stack simulation, which is based on stacks that record accesses, Sugumar and Abraham developed some more efficient algorithms using binomial trees [11]. Their new one-pass schemes are reported to outperform earlier ones by a factor of up to 5. Furthermore, they proposed a one-pass algorithm to simulate caches with varying block size. Their algorithms were implemented in Cheetah, a really fast one-pass cache simulator.

In spite of the their efficiency, one-pass simulation algorithms all have certain limitations. Since they have to track block status for all the caches in the simulation pool, the bookkeeping of each reference has to be simple, otherwise it will not make much difference from simulating the caches one by one. As a matter of fact, none of the one-pass simulations support prefetching, sub-block replacement, or multi-level caches; nor can they produce useful performance metrics such as distribution of misses.

3. Iterative cache simulation

3.1 Motivation and goals

Our attempt of evaluating cache performance originated from studies of video applications on a VLIW base video signal processor. Since the processor consists of several clusters each having its local cache, we need a simulator that can support multiprocessor caches. Due to limited inter-cluster communication bandwidth and uncommon data sharing patterns, invalidation-based protocols are preferred over update-based protocols. Moreover, we are interested in prefetching-based systems. However, no existing one-pass simulators could do the job. At first, we tried to adapt them to the new requirements, but it turned out that the modifications would be so complicated that there would not be much difference from running the simulation for each cache configuration.

It seems clear that we have to simulate all the candidates one by one. On one hand, this gives us the flexibility to deal with various cache models (e.g. multiprocessor caches, multi-level caches, caches with sub-block replacement and/or prefetching strategies, et al.) and performance metrics (e.g. write-back counts, distribution of misses) that are not supported or only partially supported by one-pass simulators. On the other hand, however, the gains might be at the cost of excessive time. Therefore our focus is on reducing simulation time.

In order to reduce simulation time, we have to reduce workload. There is always a tradeoff between accuracy and trace length. Our goal is to reduce trace length as much as possible and meanwhile keep performance metrics 100% correct.

3.2 Notations and assumptions

In the following discussions, we use a 3-tuple (*nsets*, *blksize*, *assoc*) to refer to a specific cache configuration where the cache consists of *nsets* sets and each set contains *assoc* blocks of *blksize* bytes. Therefore if *assoc*=1 it indicates a direct-mapped cache, otherwise it is an *assoc*-way set-associative cache (with LRU replacement being used). We further assume that *nsets* and *blksize* are both in powers of two.

Miss ratio is probably the most important figure among all the useful performance metrics. We will use this metric as an example in the next subsection, and then generalize our algorithms to other metrics at the end of this section.

3.3 Iterative cache simulation

As mentioned previously, we want to discard some non-interesting references in the trace after each simulation, so that next time when we simulate another cache configuration the workload could be reduced. E.g., sometimes when we go from one cache configuration *C1* to another one *C2*, all the references that hit *C1* will always hit *C2*, therefore by stripping off those references that cause hit to *C1* (which is a significant reduction), we reduce trace length yet are still able to get exact miss ratio for *C2*.

Hill and Smith found out that set-refinement property implies inclusion property in direct-mapped caches with same block size [7]. This observation facilitates the simulation of direct-mapped caches with same block size. E.g., during the simulation of cache (*256, 16, 1*), we could toss out all the references that cause hit, and later the reduced trace can be used to simulate (*512, 16, 1*) or other direct-mapped caches with more number of sets but same block size. Remember that extended inclusion property says that cache configuration (*nsets1*, *blksize*, *assoc1*) contains a subset of blocks in (*nsets2*, *blksize*, *assoc2*), provided that $nsets1 \le nsets2$ and $assoc1 \le assoc2$. So once we have the reduced trace from (*256, 16, 1*), we could also simulate set-associative caches with more than or equal to 256 sets.

Now the question is which references we should discard when simulating a set-associative cache. Of course we can use the reduced trace from direct-mapped cache for all the set-associative caches, but that is not economical as we could filter out more redundant references. Unfortunately set-refinement does not imply inclusion in set-associative caches, which means that we cannot simply throw away references that cause a hit. This is because LRU replacement may change the mapping of a missed block. E.g., suppose we have a cache configuration (*1, 1, 2*) and a reference sequence *0, 1, 0, 2, 1*; then the outcome would be, respectively, miss & place into block 0, miss & place into block 1, hit block 0, miss & replace block 1, miss & replace block 0 (Figure 3.3.1a). If we remove the references that cause hit (as we did with direct-mapped caches), the subsequent sequence *0, 1, 2, 1* would result in a hit for the last *1* (Figure 3.3.1b), since the previous miss of *2* will replace block 0, the least recently used one.

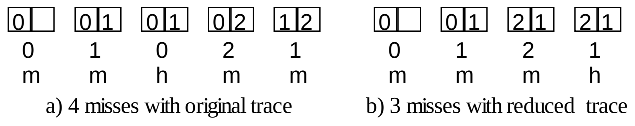
0
0 1
0 1
0 2
1 2
0 1 0 2 1
m m h m m
a) 4 misses with original trace
0
0 1
2 1
2 1
0 1 2 1
m m m h
b) 3 misses with reduced trace

Figure 3.3.1: Examples showing that set-refinement does not imply inclusion in set-associative caches

The reason for the discrepancy is that hits can change the recentness of a block. Therefore in addition to the references that cause miss, we need to save as well those hit references that alter recentness of blocks in a set. Notice that recentness does not matter at all until a miss occurs, so we only need to save at most *assoc-1* hit references before the miss reference when an *assoc*-way set-associative cache is being simulated. As shown in Algorithm 3.3.1, the overhead of this method is fairly small, but the gain in trace length reduction is significant. For many traces and cache configurations we have analyzed, after simulating 2-way set-associative cache the trace length could be cut 50%. Each time when we simulate a cache, we can strip off some redundant references, thereby accelerate further simulations of caches with more sets and/or higher associativity.

```
cache_sim( addr )
{
    /* set and blk indicate the set (in cache) and the
       block (in set) addr ended up with, no matter
       whether it is a hit or miss.
    */
    (set, blk, hit) = conventional_cache_sim( addr );
    if ( hit ) {
       set.block[blk].hit_addr = addr;
    } else {        // miss
       for ( i = 1..assoc-1 in LRU order ) do {
          // 0: LRU — assoc-1: MRU
          if ( set.block[i].hit_addr != invalid ) {
             save( set.block[i].hit_addr ); // hit reference
             set.block[i].hit_addr = invalid;
          }
       }
       save( addr );           // miss reference
    }
}
```

Algorithm 3.3.1: Cache simulation with trace reduction

So far we have only assumed same block size throughout our cache simulations. For varying block size, things become very complicated, because the inclusion property no longer holds. To deal with this problem, Wang and Baer proposed a “universal” trace [9], which is a collection of references that cause misses in each cache filter with different block sizes. In Cheetah, Sugumar and Abraham developed a one-pass algorithm based on binomial trees and tag inclusion property to simulate direct-mapped caches of same size but different block sizes [11].

	(256, 32, 3)	*(256, 64, 3)*
(512, 16, 1..3)	*(512, 32, 2..3)*	*(512, 64, 2..3)*
(1024, 16, 1..3)	*(1024, 32, 1..3)*	*(1024, 64, 1..2)*
(2048, 16, 1..3)	*(2048, 32, 1..3)*	

Table 3.3.1: Cache configurations

While Wang and Baer's algorithm applies for caches with same number of sets, Sugumar and Abraham's algorithm assumes fixed cache size. Our approach to the problem is a combination of both. It first finds out a **dominant set**, which contains the minimum cache for each block size. E.g., given the interested cache configurations in Table 3.3.1, the dominant set is { (*512, 16, 1*), (*256, 32, 1*), (*256, 64, 1*) }, as (due to inclusion property) the union of the miss trace of the three configurations will be a superset of references that cause misses in any of the caches in Table 3.3.1. Algorithm 3.3.2 shows the function to generate the universal trace using the same example. It simulates (*256, 32, 1*) and meanwhile saves the miss references for all the three configurations. Notice that cache (*512, 16, 1*) has the same size as (*256, 32, 1*), which means that the tags for both configurations are of the same size too, so it is treated differently as (*256, 64, 1*). In our implementation, this algorithm is realized using bit-vectors, so it is quite efficient.

```
univ_cache_filter( addr )
{
   (set, hit) = conventional_cache_sim( (256, 32, 1), addr );
   if ( hit ) {
      saddr = addr >> 4;  // strip off block offset
      if ( !set.fetched[saddr & 0x1] ) {
         set.fetched[saddr & 0x1] = 1;
         save( addr );        // for (512, 16, 1)
      } elsif ( last[(addr>>6) % 256] != addr>>6 ) {
         last[(addr>>6) % 256] = addr>>6;
         save( addr );        // for (256, 64, 1)
      }
   } else {                   // miss
      save( addr );           // miss reference
      clear set.fetched[ ];
   }
}
```

Algorithm 3.3.2: Generating universal trace using {(*512, 16, 1*), (*256, 32, 1*), (*256, 64, 1*)} as an example

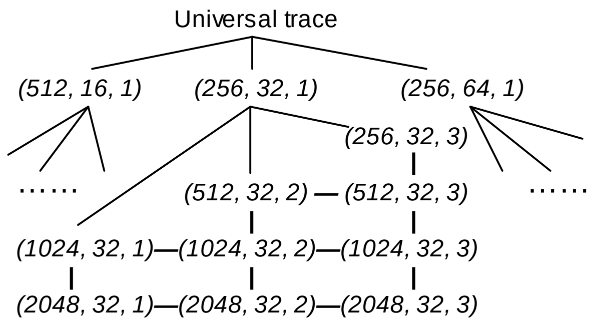

Figure 3.3.2: Lattice of some cache configurations

Once we have the universal trace, we can simulate all the cache configurations. In order to save workload to the maximum extent, we need to sort the candidate cache configurations into a special order such that following simulations can use the reduced traces generated by previous simulations. E.g., both (*512, 16, 2*) and (*1024, 16, 3*) can be simulated using the universal trace. If we simulate the former one first, then we can take advantage of the further reduced trace in the latter one. However, if the latter configuration is simulated first, we have to use the universal trace again when simulating (*512, 16, 2*). Obviously the cache configurations in our design space form a **partially ordered set**, in which the relation is defined by the inclusion property. As an example, Figure 3.3.2 depicts the lattice of a set consisting of the configurations in Table 3.3.1.

Starting from the universal trace, we can traverse the lattice in associativity first or number-of-sets first order. Depending on the trace, either one can be faster but the difference is not distinctive in our experiments.

3.4 Other performance metrics

Although miss ratio is important, there are some other interesting and useful performance metrics as well. E.g., in write-back cache systems, the frequency of write-backs would greatly affect the traffic that goes to next level of memory. Since for each cache configuration we perform a conventional simulation, it is very easy to adapt the simulation algorithm to take into account other performance metrics and/or cache architectures. In fact, for some metrics like distribution of misses, we need not change Algorithm 3.3.1 or 3.3.2 at all.

For write-back counts, we need to record references that cause write-hit. In Algorithm 3.3.1, at most *assoc-1* hit references are recorded before a miss reference. However, the write-hits that make a block dirty (and thus contribute to write-back counts) may not be among the hit references recorded, since only the most recent access to each block is kept. A trick to solve this problem is to change any read operation to write operation. Notice that this change of Algorithm 3.3.1 is minor and does not increase the trace length. However, in Algorithm 3.3.2 we have to record all the write-hits as we do not know which of them will modify which block in a set-associative cache.

As for multiprocessor caches, obviously read accesses of shared data will not cause any problem. This is because a shared read will clear the dirty flag (if set) of the block containing shared data on the exclusive processor and bring the block to the processors that issue the read, regardless of whether update- or invalidation-based protocols are used. However, things become different for a shared write. In update-based protocols, a shared write will update all the blocks that have the shared data, which has the same effect as a shared read, hence no extra processing is required for one-pass simulation algorithms. In invalidation-base protocols, however, those blocks will be invalidated by the shared write. This will cause deletion of a block in the LRU stacks used by one-pass algorithms. Tracking the propagation of a deleted block for different cache configurations is a complicated matter, so one-pass algorithms cannot handle invalidation-based protocols. Nonetheless, our method does not have such a problem, since only one cache configuration is simulated at a time. Note that update or invalidation happens only when a miss occurs. Because all the miss references are saved anyway, Algorithm 3.3.1 and Algorithm 3.3.2 do not need any modification.

4. Results

Since we are interested in video applications and the traces of several such programs are readily available, we use them as examples. The outputs (performance metrics) of our simulations are correct when compared with results from other methods. Discussions of the miss ratios or the impact of cache configurations on video applications are beyond the scope of this paper, but they can be found in our other work [12]. In this section we will concentrate on the execution time. The generation of original traces and our trace-driven cache simulations are done on an SGI Power Challenge workstation which has four 194 MHz MIPS R-10000 microprocessors.

Applications	Original	First	Last
H.263 encoder	329,872 K	5,286 K	78 K
H.263 decoder	47,071 K	623 K	4 K
MPEG-2 encoder	113,820 K	5,704 K	17 K
MPEG-2 decoder	2,550 K	53 K	3 K
MPEG-4 encoder	358,393 K	7,709 K	1,875 K
MPEG-4 decoder	274,889 K	9,791 K	3,772 K

Table 4.1: Trace length of six video applications

Table 4.1 shows the trace lengths of original data references. To highlight the reduction of disk space, we also show in the table the trace lengths after the first and last simulations. We compared the simulation time of our method with Cheetah, which is a great one-pass cache simulator in terms of speed but lacks flexibility. It only reports miss ratio and it does not support complex cache models (e.g. caches using write-back, prefetching, and/or sub-block replacement). Moreover, it can only simulate caches with the same block size or direct-mapped caches with the same cache size. For these reasons, we only measured miss ratios for (*256..2048, 32, 1..4*) (i.e. fixed block size — 32 bytes) and { (*256, 128, 1*), (*512, 64, 1*), (*1024, 32, 1*), (*2048, 16, 1*) } (i.e. fixed cache size — 32 Kbytes). The results are shown in Table 4.2 and Table 4.3 respectively. We also compared our iterative method with traditional trace length reduction technique, in which the reduced trace generated during the simulation of (*256, 32, 1*) is used for all the other configurations (i.e. "multi-run"). A speedup of 10-20% of iterative method over multi-run is observed in Table 4.2. As we can also see, our method outperforms Cheetah for caches with the same block size (Table 4.2); but for fixed-size direct-mapped caches, it could be 50% slower (Table 4.3), because for each configuration we have to use the universal trace.

Applications	Cheetah	Multi-run	Iterative
H.263 encoder	164.12 s	192.82 s	150.12 s
H.263 decoder	14.86 s	17.33 s	14.71 s
MPEG-2 encoder	66.57 s	71.77 s	58.76 s
MPEG-2 decoder	0.87 s	0.98 s	0.77 s
MPEG-4 encoder	190.22 s	207.39 s	185.04 s
MPEG-4 decoder	169.33 s	171.27 s	150.97 s

Table 4.2: Simulation time with fixed block size

Applications	Cheetah	Iterative
H.263 encoder	83.96 s	146.64 s
H.263 decoder	11.19 s	11.81 s
MPEG-2 encoder	45.62 s	51.72 s
MPEG-2 decoder	0.55 s	0.69 s
MPEG-4 encoder	146.51 s	158.47 s
MPEG-4 decoder	112.52 s	120.09 s

Table 4.3: Simulation time with fixed cache size

5. Conclusions

In this paper, we presented a so-called iterative cache simulation method, to trace-driven simulate a set of cache configurations accurately and quickly. In our approach, we sort candidate cache configurations in such an order that after each simulation we can reduce trace length and thus speedup following simulations. Compared with other cache simulators, our method features following:

1. It supports a wide range of performance metrics, including miss ratio, write-back counts, bus traffic, et al.
2. Unlike trace sampling techniques which cannot guarantee accuracy, the results produced by our simulator are 100% precise.
3. It is flexible in terms of supporting various cache models such as uniprocessor and multiprocessor caches, multi-level caches, and caches with sub-block replacement and prefetching.
4. Empirically it has similar speed as Cheetah, the fastest simulator ever reported.

References

[1] Yanbing Li and Wayne Wolf, "Hardware/software co-synthesis with memory hierarchies", *Proc. Int'l Conf. on Computer Aided Design*, pp. 430-436, Nov. 1998.

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

[3] Alan J. Smith, "Two methods for the efficient analysis of memory address trace data", *IEEE Trans. on Software Engineering*, 3(1), pp. 94-101, Jan. 1977.

[4] Thomas R. Puzak, "*Analysis of Cache Replacement Algorithms*", Ph.D. Thesis, Univ. of Massachusetts, Amherst, Feb. 1985.

[5] Anant Agarwal and Minor Huffman, "Blocking: exploiting spatial locality for trace compaction", *Proc. 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 48-57, May 1990.

[6] Subhasis Laha, Janak H. Patel , and Ravishankar K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems", *IEEE Trans. on Computers*, 37(11), pp. 1325-1336, Nov. 1988.

[7] Mark D. Hill and Alan J. Smith, "Evaluating associativity in CPU caches", *IEEE Trans. on Computers*, *38(12)*, pp. 1612-1630, Dec. 1989.

[8] James G. Thompson and Alan J. Smith, "Efficient (stack) algorithms for analysis of write-back and sector memories", *ACM Trans. on Computer Systems*, 7(1), pp. 78-117, Feb. 1989.

[9] Wen-Hann Wang and Jean-Loup Baer, "Efficient trace-driven simulation methods for cache performance analysis", *Proc. 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 27-36, May 1990.

[10] Yuguang Wu and Richard Muntz, "Stack evaluation of arbitrary set-associative multiprocessor caches", *IEEE Trans. on Parallel and Distributed Systems*, 6(9), pp. 930-942, Sep. 1995.

[11] Rabin A. Sugumar and Santosh G. Abraham, "Efficient simulation of multiple cache configurations using binomial trees", *Technical Report CSE-TR-111-91*, CSE Division, Univ. of Michigan, 1991.

[12] Zhao Wu and Wayne Wolf, "Study of cache system in video signal processors", *Proc. IEEE Workshop on Signal Processing Systems*, pp. 23-32, Oct. 1998.