DEW: A Fast Level 1 Cache Simulation Approach for Embedded Processors with FIFO Replacement Policy

Mohammad Shihabul Haque  Jørgen Pedersen  Andhi Janapsaty  Sri Parameswaran
University of New South Wales, Sydney, Australia
{mhaque, jorgent, andhij, srivegan}@cse.unsw.edu.au

ABSTRACT
Increasing the speed of cache simulation to obtain hit/miss rates enables performance estimation, cache exploration for embedded systems and energy estimation. Previously, such simulations, particularly exact approaches, have been exclusively for caches which utilize the least recently used (LRU) replacement policy. In this paper, we propose a new, fast and exact cache simulation method for the First In First Out (FIFO) replacement policy. This method, called DEW, is able to simulate multiple level 1 cache configurations (different set sizes, associativities, and block sizes) with FIFO replacement policy. DEW utilizes a binomial tree based representation of cache configurations and a novel searching method to speed up simulation over single cache simulators like Dinero IV. Depending on different cache block sizes and benchmark applications, DEW operates around 8 to 40 times faster than Dinero IV. Dinero IV compares 2.17 to 19.42 times more cache ways than DEW to determine accurate miss rates.

1. INTRODUCTION
Cache memories have been used to effectively reduce the ever increasing speed gap between the main memory and the processor. Utilizing data and instruction caches in computing systems improves performance while reducing energy consumption.

A processor based embedded system, where an application or a class of applications is repeatedly executed, can be customized by the adroit selection of a suitable cache. Multiple studies [5, 8, 13, 18] have found that the correct combination of different cache parameters, such as the cache size, number of cache sets (set size), associativity, cache block size (also known as cache line size), etc. can reduce the energy consumption and increase the overall system performance significantly. Application specific processor design platforms such as Tensilica’s Xtensa [2, 15] allows the cache to be customized for the processor to meet tighter energy, performance and cost constraints. A cache system which is too large will unnecessarily consume power and increase access time, while a cache system too small will thrash, reducing performance.

Due to the erratic nature of caches, there is no known way of accurately determining hit and miss rates without simulating an application’s trace of memory requests. To simulate the trace on caches with hundreds of differing cache parameters can take several months and is simply not feasible. Therefore, several studies have endeavored to speed up simulation of cache memories. Among the simulation methods, some approaches simulate caches with all the possible combinations of cache parameters under consideration extensively to maintain reliability (i.e., exact values of hits and misses). These are called ‘Exact Approaches’⁷. One of the widely used exact approach based single processor cache simulation tool is Dinero IV [7], designed by Jan Elder and Mark Hill. Dinero IV can simulate only a single combination of cache parameters at a time. Among the exact approaches, some approaches [13, 20] are able to simulate multiple combination of cache parameters in a single pass directly over an application trace. These approaches mainly depend upon cache inclusion properties to speed up simulation. However, caches with the FIFO (or round robin) policy do not exhibit inclusion properties. Therefore, there has been no known work which attempts to speed up the simulation of multiple caches which implement the FIFO replacement policy.

As FIFO replacement is inexpensive to implement in the hardware, FIFO is a popular choice for level 1 cache in the embedded processors (i.e., Xtensa LX2 processor [3] and Intel XScale processors [11]). Besides that, previous studies [4] have shown that for L1 cache (especially, data cache), both FIFO and LRU have their own advantages. Therefore, in our research, we have decided to extend the simulation approach for FIFO replacement policy. In our research, we have analyzed the features of FIFO replacement policy that prevent us from establishing fast simulation properties when all the caches under simulation use the FIFO replacement policy. We have studied the exact simulation methods, especially Janapsaty’s method with the proposed enhancements in the CRCB algorithm [13, 20], to determine how the inclusion properties benefit simulators. Resulting from our findings, we propose a new simulation strategy “Direct Explorer Wave” (DEW) to speed up simulation of multiple combination of cache parameters with FIFO replacement policy in a single pass directly over an application trace.

The rest of the paper is structured as follows. Section 2 presents related works, Section 3 presents the background of our research, Section 4 describes our DEW simulation approach, Section 5 describes the experimental setup and presents the results found for mediabench applications; and section 6 concludes the paper.

2. RELATED WORK
Cache performance evaluation has been studied extensively for a long time to find the optimal combination of cache parameters for level 1 cache in embedded systems. The methods of cache evaluation can be broadly categorized in two: estimation and simulation dependent. Estimation approaches [8, 10, 17, 21] depend on heuristics, are fast to compute, but are limited in their accuracy. Simulation based approaches [7, 12, 13] usually produce error free results of cache hits and misses. However, they take a longer time than estimation approaches to execute.

Several techniques are used to make simulation of application traces faster. One such technique is fractional simulation [12, 16], which allows the simulation of a section of the trace, and obtains results at the cost of accuracy. Another technique simulates the trace for a number of cache configurations (different combination of cache parameters) simultaneously, and produces exact results. These concurrent simulations use the knowledge of cache behavior between configurations to speed up simulation considerably. For example, if a hit occurs in a cache with four sets, it is guaranteed to be a hit on a cache with eight sets, provided that both of the caches use the Least Recently Used (LRU) replacement policy, and have equal associativity and block size.

Due to the reliability, many methods have been proposed to improve the speed of exact, concurrent, simulation based cache evaluation approaches. In 1989, Hill et al. in [11] studied the effect of associativity in caches. They introduced a forest simulation technique to

⁷ Advisor
simulate alternate direct mapped caches quickly. Another technique used was the all-associativity methodology, based on the “Stack” algorithm described by Gecsei et al. in [9], for simulating alternate direct mapped caches, fully-associative caches and set associative caches. Hill et al. showed that for alternate direct mapped caches, forest simulation strategy is faster than the all-associativity methodology. In 1995, Sugumar et al. [19] proposed a binomial tree dependent cache simulation methodology to improve methods described in [11]. Sugumar’s method had a time complexity of $O((\log_2(X) \times A)$ for searching, where $X$ and $A$ are size and associativity of the cache respectively. Time complexity of maintaining the tree was $O((\log_2(X) \times A)$. Sugumar’s method was applicable only for LRU replacement policy. Due to its flexibility, Sugumar’s method promoted the use of binomial tree in simulation of multiple cache configurations in a single pass, taking as its input an application trace. Researchers have continued the use of binomial tree to speed up simulation though the focus has remained only on LRU replacement policy. In 2004, Li et al. [16] proposed an improvement to Sugumar’s methodology by introducing a compression method to reduce simulation time. The authors of [16] stated that their method can be modified to accommodate the FIFO replacement policy; however, no modification plan for the FIFO replacement policy was given.

In 2006, Janapsaty et al. [13] proposed a technique by utilizing several LRU based cache inclusion properties and a binomial tree structure. Janapsaty’s top-down tree traversal based simulation strategy helped to speed up simulation of multiple cache configurations by reading the application trace only once. Janapsaty’s searching approach, inside a cache set, took advantage of temporal locality to speed up simulation, as memory address tags were searched according to their last access time. Therefore, Janapsaty’s method had a shorter simulation time than previously proposed solutions. The cache properties and techniques used in Janapsaty’s method was exclusive for the LRU replacement policy. Janapsaty’s method had a fixed time complexity of $O(\log_2(X) \times A)$ for searching data or instructions inside the caches under simulation, where $X$ and $A$ are maximum cache set size and maximum associativity respectively. Time complexity for updating the data structure was $O(\log_2(X))$. In 2009, Tojo et al. [20] proposed two enhancements to Janapsaty’s method in what they called the CRCB algorithm. These pruning based proposals made the simulation even faster by reducing the number of addresses to be examined. The findings of CRCB are also true for FIFO replacement policy; however, the simulation technique was exclusively proposed for the LRU replacement policy.

2.1 Contributions and limitations

1. In this paper, we have presented a new simulation strategy “DEW” to simulate multiple level 1 cache configurations of varying set sizes with the FIFO replacement policy by passing over an application trace only once.

2. A novel data structure based on binomial trees and utilizing “wave pointers” has been proposed to enable fast simulation.

3. A search methodology for the above data structure has been proposed, which eliminates unnecessary tag comparisons.

The limitation of DEW is that it is optimized only for the simulation of the FIFO replacement policy. It can simulate caches with the LRU replacement policy, but will typically be slower than Janapsaty’s method [13] and the CRCB algorithm [20], which are optimized only for the LRU policy.

3. CACHE PARAMETERS EXPLORATION METHODOLOGY

Cache configurations are mainly parameterized using cache set size (S), associativity (A) and cache block size (B). Cache size (T) is the total number of bits that can be stored in the cache. Cache set size (S) is the total number of sets in a set associative cache. The number of ways to place data inside a set of a set associative cache is called the associativity(A). Cache block size(B), also known as cache line size, is the minimum amount of data that can be stored in a cache. Therefore, $T = S \times B \times A$.

In DEW, we perform simulation on the cache parameters to estimate the number of cache misses that would occur for a given collection of cache configurations. In DEW, we optimize the run time of simulation by replacing multiple readings of large program traces with a single reading. Simulating multiple cache configurations simultaneously and reducing search complexity inside a cache configuration. This is possible due to the data structure we have used and the decisions we can make depending on the data structure. In the following subsections, we are going to discuss the data structure used in DEW and the properties that can be used due to the special data structure.

3.1 Data structure used in DEW

To perform simulation of an application trace on cache parameters, a forest of binomial trees are created in DEW for each cache block size and associativity. Each tree represents a group of caches with configurations of same block size and associativity. The root node represents a set size of one (fully associative). The second level of the tree contains two nodes, and represents cache with two sets, with each node storing the contents of one set. Subsequent child levels continue to double the set size. Simulation may begin at a lower level of the tree.

Figure 1 depicts a case starting with a set size of two (see top nodes 0 and 1). The first node on the left, marked ‘0’, refers to cache set with index 0. And the second node, marked ‘1’, refers to cache set 1. At the second level of the two trees, there are a total of four nodes marked ‘00’, ‘10’, ‘01’ and ‘11’. Thus the second level will represent a cache with $S = 4$, and the numbering within the nodes will represent the respective cache sets as shown in Figure 1. Similarly, the third level (depicted as the bottom level in Figure 1), will represent a cache with eight sets. For caches with larger set numbers, the tree is further expanded with a greater number of levels.

To store tags, each simulation tree node is associated with a list. Each entry inside the tag list represents a cache way. Tag list entries are picked in the round robin fashion for tag replacement. Each address tag inside a tag list keeps a pointer to the cache way of a set in the child node that holds the tag for the same address (the child level shares the same cache associativity and block size, but has twice as many nodes). We will call this special knowledge about tag location the “wave pointer”. Inside a tree node, whenever the tag of a list entry is replaced by a new tag, the old tag and its wave pointer is stored in a special entry, “Most Recently Evicted (MRE)”, associated with the tree node. The newly placed tag is also stored in another special entry, “Most Recently Accessed (MRA)”, associated with the tree node.

During evaluation of a tree node(cache set), whenever a tag is found(on hit) or inserted(on miss) into the $V^{th}$ tag list entry of any cache set, the tag list entry is called the “Matching entry”. The wave pointer of the matching entry is left untouched on a hit, made empty on a miss, or replaced by the wave pointer of the MRE tag's wave
pointer if the the MRE tag is the requested tag. On any access, the wave pointer of the matching entry in the parent tree level is updated with \( V \). Thus the wave pointers will form a flow of links between matching entries moving down the tree.

An example simulation tree has been presented in Figure 2. The first tree level represents a cache configuration with set size one, associativity four and block size of one byte. Similarly, the second tree level represents a cache configuration with associativity four and block size one byte; however, set size is two. The cache sets are holding the tags for the byte addressable memory adddress requests presented in the application trace in Figure 2. The numbers on the curved arrows are the wave pointers of the tag list entries at the curved arrow’s bases. The tag for the address 1011 is stored in the second cache way of the top tree level’s (level 1) cache set. The same address tag is stored inside the second cache way of cache set 1 of the bottom tree level. Therefore, in the top level, the second entry of the tag list will store wave pointer 2. Note that the matching entry of the top level does not store the cache set number of the bottom level. This is because, the memory address itself determines the child level cache set needs to be checked. For the application trace of Figure 2, in the top level of the simulation tree, the MRA tag is 1011 and the MRE tag is 1100.

![Figure 2: A simulation tree for DEW](image)

### 3.2 Properties of the data structure in DEW

The data structure used in DEW enables the simulation process to utilize several properties to speed up simulation. It helps to reduce simulation time in the following way:

**Property 1: Use of Binomial tree for simulation on cache parameters**

The binomial tree representation enables the simulation process to simulate multiple cache configurations by reading the application trace only once. This feature of the binomial tree based simulation can be explained with the aid of a simple example. Let us suppose that the memory address ‘10001010’ has to be simulated. Such an address will store its content in index 0 in the cache with two sets and in index 10 in the cache with four sets (assuming byte addressable memory and \( B = 1 \) byte) in the forest of Figure 1. Thus with the aid of the tree structure, by first reading the last bit of the memory address, we can store the address in the index 0 of the two-set cache. Then, the link from that node can be followed by reading the “penultimate” bit of the address. Since the penultimate bit is a 1 in the example, the node 10, to which there is a link from the node 0, can be quickly simulated without searching the trees for appropriate cache set. Due to this strategic tree structure, a large number of caches with differing sizes can be simulated simultaneously, with minimum amount of searching.

**Property 2: Storing MRA tag separately**

According to the observations of the CRCB algorithms [20], processing of the MRA tag of a cache set cannot be a miss regardless of the size, associativity, replacement policy and block size of the cache. During simulation in DEW, processor requested addresses are read from the selected application’s trace file one at a time and sent to each forest. Inside a forest, a tree is selected and simulated using the procedure we have described previously. As a result, the MRA tag of a cache set will be the MRA tags of the appropriate cache sets in the bigger set sized caches. Thus, whenever the MRA tag of a cache set is requested, there is no need to simulate the address tag in the bigger set sized caches in the DEW forest. As each DEW simulation tree node stores the MRA tag (see Figure 2), it is possible to utilize this observation of the CRCB algorithms [20].

**Property 3: Use of wave pointer to determine hit or miss**

By storing the wave pointer, each memory address tag inside a cache set knows the only possible location of the tag for the same address in the child level of a tree. Therefore, whenever a hit occurs in a cache configuration, the tag list entry in the child level pointed by the current level’s matching entry wave pointer should be checked for the requested address tag. If the two entries (at either end of the wave pointer) do not match, DEW declares it a miss without having to search the other entries in the child level. This is because, whenever a tag is placed in a tag list entry, the parent level’s matching entry wave pointer is informed about the location. Therefore, if the tag is present in the child level, the parent level’s matching entry for the same address tag will accurately point to the tag location in the child level. However, if a child level tag is replaced, the wave pointer of its parent level’s matching entry cannot be updated. Therefore, some wave pointers may point to incorrect tag locations in the child level in a DEW simulation tree. However, an incorrect wave pointer guarantees that the tag is not available elsewhere in the child cache configuration (tree level) as the replacement is performed in the round robin fashion. Therefore, wave pointers create a directed flow of matching tags from the top tree level to the bottom; hence, helping to avoid unnecessary tag comparison in the tag list.

**Property 4: Storing MRE tag separately**

Due to temporal locality, the most probable tag to be requested again, amongst the evicted tags, will be the MRE tag. Thus, DEW stores the MRE tag of a cache set and its wave pointer in a special entry associated with each tree node (cache set). If the requested tag is found to be the MRE tag in any simulation tree node, then a cache miss is indicated. Therefore, storing the evicted tag can save simulation time. The wave pointer of the MRE tag can save time to find a tag in the child level in a simulation tree. As in Property 3, the wave pointer of the MRE tag will become invalid if the child level matching entry is replaced. However, an incorrect wave pointer from the MRE tag still guarantees that the tag is not available elsewhere in the child level in the simulation tree.

The four features mentioned above help DEW find the total number of misses of a collection of cache configurations accurately without simulating each and every cache configuration separately, and reduces the total simulation time by reducing the total number of tag comparisons.

### 4. DEW SIMULATION APPROACH

DEW is a strategy to speed up simulation of multiple level 1 cache memories with FIFO replacement policy in a single pass directly over an application trace. To record total number of cache misses for different cache configurations, DEW keeps a table, indexed with the cache configuration parameters: set size, associativity and cache block size. Size of the table is dependent on the total number of cache configurations considered for simulation.

In a forest, simulation trees are simulated in the Top-Down fashion. In other words, for each memory address request, simulation starts in the smallest cache configuration of a forest and finishes in the biggest one. To evaluate a tree level for a memory address request, a cache set (tree node) is selected using the procedure we have described earlier in Section 3.1.

Figure 3 illustrates the above flow to show how an address request is evaluated in a simulation tree level. In a tree node, Address evaluation starts from the start point of Figure 3. Depending on the tests of the flow diagram, DEW either decides a hit or a miss. On a miss and hit, the functions Handle_miss() and Handle_hit() are called respectively to perform necessary actions. The functions Handle_miss() and Handle_hit() are described in Algorithm 2 and Algorithm 1 respectively. DEW finishes evaluation of a tree node either by going
to the child level cache set or the end point of Figure 3. In the child level cache set, evaluation starts from Figure 3’s start point. A textual description of the flow of Figure 3 is given below:

1. Inside each tree node, the MRA tag is checked first for the requested address tag. If the requested address is found, DEW skips evaluation of the address in the bigger cache configurations by going to step 9. Else, DEW continues to step 2.
2. DEW checks whether the parent level’s matching entry’s wave pointer is empty or not. If wave pointer is not empty, DEW goes to step 3, else to step 6.
3. DEW checks whether the parent level’s matching entry’s wave pointer points to the correct tag or not. If the pointed tag is correct, DEW moves to step 4, else to step 5.
4. DEW performs the operations for a hit. The current tree node’s MRA tag is updated to the requested tag. Parent level’s matching entry wave pointer will be updated. DEW will then use the wave pointer of the current level’s matching entry to continue evaluation in the next level by continuing from step 8.
5. DEW records a miss. The miss counter for the current cache configuration will be increased by one. If the current level’s MRE tag is the requested address tag, DEW will exchange the wave pointer and the least recently inserted tag container entry’s tag with the MRE entry’s tag and wave pointer; else, the least recently inserted tag will be replaced by the requested tag, the wave pointer for the newly inserted tag will be empty and the MRE tag and its wave pointer will be stored. Parent level’s matching entry wave pointer will be updated. MRA tag of the current cache set will be updated with the newly inserted tag. After that evaluation of the requested address will be performed in the next level with the matching entry wave pointer by continuing from step 8.
6. DEW will check whether the current level’s MRE tag is the requested address tag or not. If MRE is the requested address tag, DEW goes to step 5, else to step 7.
7. DEW performs a search for the tag in the tag list entries one by one, starting from the first entry (Way 1) of the list. We will call each of the comparisons performed by the search “Tag comparison”. If the tag is found in any tag list entry, DEW goes to step 4; otherwise, DEW goes to step 5.
8. Operation for the current tree level is complete. If the level is the final level of the tree, operation for the tree ends via step 9. Otherwise, the appropriate node of the child level is evaluated from step 1.
9. Operation for the current request is complete. Operation can continue on the next trace request.

We now present three examples of the DEW algorithm working from the trace in Figure 2 in each case. Figure 4 shows the new tree after execution of these three examples.

1. **Access request to address 1011**: Level 1 will be selected first. As the MRA tag of the current level’s cache set is the requested tag, DEW will treat this as a hit in this level and subsequent levels. Evaluation for this address ends.

2. **Access request to address 0110**: In level 1, cache ways will be checked sequentially as 0110 is neither the MRA tag nor the MRE. On the hit in the third cache way, DEW will make 0110 the MRA tag of level 1 and start evaluation in level 2’s set 0. As the MRA tag is different, way 3 of set 0 will be checked for the tag because of the parent level’s matching entry wave pointer. As the way 3 of set 0 gets a hit, DEW will update level 1’s way 3’s wave pointer by 3, make 0110’s tag the MRA tag of level 2 and fetch the next request for simulation.

3. **Access request to address 1100**: A match is found in the level 1’s MRE entry when checked after the MRA tag. In this case, DEW will place the tag of 1100 in the tag list. Level 1’s MRE, MRA and miss counter will be updated. After that, the same address tag will be checked in the way 2 of cache set 0 in level 2 due to wave pointer. As the evaluation in level 2 will be a hit, wave pointer of way 3 in level 1 will be updated by 2 besides the update of set 0’s MRA.

**5. EXPERIMENTAL PROCEDURE AND RESULTS**

With the implementation described above, DEW can reduce total simulation time compared to the state of the art cache simulation tool Dinero IV [7]. We have implemented DEW using C++. We have compiled and simulated programs from Mediabench [14] with “SimpleScalar/PISA 3.0d” [6]. Program traces were generated by SimpleScalar and fed into both Dinero IV and DEW. We have verified hit and miss rates of DEW by comparing with Dinero IV and found that they are exactly the same. Simulations were performed on a machine with dual core Opteron@2GHz processor and 8GB of main memory.

In our implementation of DEW, each tag list is an array and each entry is used to hold a tag (32 bits) and integer wave pointer (32 bits). In total, each tag list entry needs to store 64 bits. In the simulation tree, each node stores the MRA tag (32 bits), MRE tag (32 bits) and wave pointer for the MRE tag (32 bits). Therefore, per tree node or cache set, (96 + (64 × A)) bits are needed, where A is associativity. Thus, per tree level or cache configuration, (S × (96 + (64 × A))) bits are needed, where S is the number of sets.

Table 1 shows how the 525 configurations we calculated data for were derived.

| Cache Set Size | Cache Block Size | Associativity |
|----------------|-----------------|--------------|
| 2^i            | 2^i             | 2^i          |
| w ≤ i ≤ 14     | w ≤ i ≤ 6       | w ≤ i ≤ 4    |

**Figure 3: An address request simulation flow diagram for DEW**

**Figure 4: Simulation tree of DEW after new tag insertion**

We have simulated cache sizes from 1 byte to 16MB, some of which may be impractical in embedded systems, to have only one tree per forest, and to follow the same experimental methodology.
Algorithm 1 Function Handle_join()
1: n= position of the cache way which holds the requested tag;
2: MRA tag of current cache set= Requested tag;
3: Parent node’s Matching entry’s wave pointer=n;
4: Matching entry location=n;

Algorithm 2 Function Handle_jmiss()
1: MRA tag of the current cache set= Requested tag;
2: Increase miss counter for the current cache configuration;
3: n= position of the cache way which holds the least recently inserted tag;
4: If The MRA tag of the current cache set is the requested tag then
5: Exchange current cache set’s n-th cache way’s tag and wave pointer
   with the tag and wave pointer of the MRE entry;
6: else
7: Replace current cache set’s n-th cache way’s tag and wave pointer
   with the requested tag and “empty”;
8: Update the MRE tag of the current cache set and its wave pointer
   with the newly evicted tag and its wave pointer;
9: endif;
10: Parent node’s Matching entry’s wave pointer=n;
11: Matching entry location=n;

used in CRCB algorithms [20].

Six Mediabench applications were used to verify the simulators. These are: JPEG encode, JPEG decode, G721 encode, G721 decode,
MPEG2 encode and MPEG2 decode. The numbers of memory address
requests have been presented in Table 2 for each of the used

| Application            | Number of requests |
|-----------------------|--------------------|
| Jpeg encode(CJPEG)    | 25,680,911         |
| Jpeg decode(DJPEG)    | 6,761,858          |
| G721 encode(G721Enc)  | 1,384,995,889      |
| G721 decode(G721Dec)  | 1,364,757,880      |
| Mpeg2 encode(MPEG2Enc)| 2,478,651,156      |
| Mpeg2 decode(MPEG2Dec)| 1,111,324,090      |

Table 2: Trace files used for simulation

Table 3 presents results comparing the DEW simulation approach
to Diner IV. [1] Column 1 lists the applications being simulated. Columns 2 shows block size. Columns 3 to 8 show simulation time and columns 9 to 14 show the number of tag comparisons performed by DEW and Diner IV for different cache associativity. E.g., columns 3 and 4 provide simulation time for DEW and Diner IV respectively to simulate direct mapped (1-way) and 4-way set associativity. Direct mapped cache results are used in both cases as DEW automatically simulates it while simulating any other associativity. Note that Diner IV is much faster than Diner IV in every case. On average, DEW operates 18 times as fast as Diner IV. This is due to the significant reduction in tag comparisons.

Figure 5 shows speedup of DEW over Diner IV based on simulation time. Speedup is calculated as the ratio of simulation times. It shows that DEW can run up to 40 times faster than Diner IV for Jpeg decode, associativity 8 and block size 64 bytes. In the worst case, DEW’s run time is still 9 times faster than Diner IV which was recorded for MPEG2 decode, associativity 4 and cache block size 4 bytes.

Figure 6 shows the percentage reduction of the total number of tag comparisons of DEW over Diner IV. From Figure 6, it can be seen that DEW can reduce the total number of tag comparisons by 54.9% to 94.9% compared to Diner IV. DEW reduces 92.97% tag comparisons compared to Diner IV for JPEG Decode, block size of 64 byte and associativity 4; however, when block size is 4 byte, DEW reduces 70.19% tag comparisons. From Figure 5, it can be seen that speed up of DEW over Diner IV for these two cases are 39 times and 23 times respectively. The correlation of Figure 5 and Figure 6 illustrates that reduction of tag comparisons helps DEW to reduce total simulation time.

It should be noted that Diner IV collects different types of information about a cache, such as the number of compulsory misses, number of demand fetches, etc., in addition to cache hit and miss rates. As Diner IV can simulate only one configuration at a time, to simulate each cache configuration, Diner IV needs to build the storage for the tags and other information. Maintaining the large information set increases the total simulation time for Diner IV.

Table 4 shows the effectiveness of each optimization property used in DEW compared to individual simulation of each cache configuration in a simulation forest of DEW without any of the properties described in Section 3.2. In this table all the results are for cache with block size of 4 bytes. Column 1 lists the applications being simulated. Column 2 shows number of tree nodes needed to be evaluated when only Property 1 (i.e., Binomial tree representation) is used in DEW. This is the worst case number of evaluations for any algorithm. Column 3 shows the total number of simulation tree nodes actually evaluated in DEW using all the four properties of Section 3.2. Column 4 shows how many of the evaluations of Column 3 found the tag in the MRA entry (Property 2); hence avoiding further evaluation of larger set sizes. These three results are associativity independent. Column 5 to 7 and 10 to show, for 4-way and 8-way associativity (including 1-way) respectively, how many times a tag list of a cache set is searched for a requested tag as well as the number of times DEW’s properties that avoid searches occurred. For example, column 5 shows total number of tag list searches performed in DEW for associativity 4. Column 6 and 7 show the number of situations, for associativity 4, when a tag list searching is avoided due to hit or miss determined by wave pointer (Property 3) or MRE entry (Property 4).

From Table 4, it can be seen that the number of node evaluations and the number of situations when a cache set is searched is significantly smaller when all the properties of DEW are used. The first line of Table 4 can be interpreted as follows. For the JPEG Encode application, without any optimization, the number of node evaluations would be 770.43 million. However, DEW reduced the total evaluations performed to 140.66 million. This large reduction is due to the property 2 (MRA), which occurred 23.18 million times. Among these 140.66 million evaluations, cache set searching has been performed only in 83 million cases for associativity of 4. The reductions arise from the use of properties 3 (Wave) and 4 (MRE) 25.47 million times and 10.24 million times respectively. Therefore, it is evident that the DEW properties are effectively helping to reduce simulation time significantly.

When a tag is available in all the cache configurations in a simulation forest, time complexity for DEW’s simulation is $O(\log_2(X))$, where $X$ is the maximum cache set size in the search space. If the tag was requested in the previous step, DEW needs only one test. For compulsory misses, time complexity for DEW’s simulation is $O(\log_2(X) \times A)$ at best. Diner IV’s time complexity for simulation of a tag is $O(\log_2(X) \times A)$ for all the cases.

Therefore, considering all the results and complexities, we can say that DEW shows the fastest performance compared to any other method proposed so far for simulation of level 1 cache with the FIFO replacement policy.

6. CONCLUSION

In this paper, we have presented a fast cache simulator, DEW, that can simulate multiple level 1 cache configurations with FIFO replacement policy in a single pass directly over an application trace. Utilizing the features of a binomial tree representation of cache configurations, DEW is able to reduce the total number of comparisons by up to 94.9% compared to Diner IV. As a result, DEW can be almost 40 times faster than Diner IV. Even in the worst case, DEW is almost 8 times faster than Diner IV.

7. REFERENCES

[1] Intel xscale microprocessor data book. www.intel.com.
[2] Xtensa processor. http://www.tensilica.com.
[3] Xtensa i2x data book. www.tensilica.com. 3/2007.
[4] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpi2000 benchmark suite. In ACM SE 42: Proceedings of the 42nd annual Southeast regional conference, pages 267–272, New York, NY, USA, 2004. ACM.

This research was supported under Australian Research Council’s Discovery Projects funding scheme (Project Number DP0986091).

1 Due to space limitations, only limited results are presented.
Table 3: Comparison between Dinero IV and DEW showing simulation time and total number of tag comparisons

| Application | Unoptimized evaluations | DEW evaluations | MRA count (Property 2) | Associativity 1 & 4 | Associativity 1 & 8 |
|-------------|------------------------|----------------|------------------------|---------------------|---------------------|
| JPEG enc.   | 770.43                 | 140.66         | 23.18                  | 83.00               | 25.47               |
| JPEG dec.   | 228.52                 | 46.92          | 7.31                   | 28.46               | 8.62                |
| GT21 enc.   | 4,649.99               | 975.85         | 140.30                 | 625.12              | 165.45              |
| GT21 dec.   | 4,645.69               | 998.35         | 141.07                 | 656.09              | 179.16              |
| MPE2G enc.  | 112,105.54             | 28,875.48      | 3,582.20               | 19,213.83           | 4,851.68            |
| MPE2G dec.  | 42,343.92              | 11,685.94      | 1,394.73               | 7,640.57            | 1,964.88            |

Table 4: Effectiveness of properties used in DEW (all results in millions)

| Application | CPEG | DEPEG | GT21_Enc | GT21_Dec | MPE2G_Enc | MPE2G_Dec |
|-------------|------|-------|----------|----------|-----------|-----------|
| Block Size  | Asso  | Asso  | Asso     | Asso     | Asso      | Asso      |
| 100         | 0.10  | 0.10  | 0.10     | 0.10     | 0.10      | 0.10      |

Figure 6: Reduction of tag comparison in DEW

Symposium on Microarchitecture, pages 330–335, 1997.

Figure 5: Speed up of DEW over Dinero IV

15. S. Leibson and J. Marsea. Flix: Fast relief for performance-hungry embedded applications. Technical report, Temslica Inc., 2005.
16. X. Li, H. S. Negi, T. Mitra, and A. Roychoudhury. Design space exploration of caches using compressed traces. In ICS ’04: Proceedings of the 16th annual international conference on Supercomputing, pages 116–125, New York, NY, USA, 2004. ACM.
17. J. J. Peiper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim. High level cache simulation for heterogeneous multiprocessors. In DAC ’04: Proceedings of the 41st annual conference on Design automation, pages 287–292, New York, NY, USA, 2004. ACM.
18. D. Ponomarev, G. Kucuk, and K. Ghose. Accupower: An accurate power estimation tool for superscalar microprocessors. In DATE ’02: Proceedings of the conferen on Design, automation and test in Europe, page 124, Washington, DC, USA, 2002. IEEE Computer Society.
19. R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binary trees. ACM Trans. Comput. Syst., 11(1):32–56, 1995.
20. N. Taji, N. Togawa, M. Yanagisawa, and T. Ohtsuki. Exact and fast 11 cache simulation for embedded systems. In ASP-DAC ’09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation, pages 817–822, Piscataway, NJ, USA, 2009. IEEE Press.
21. X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. ACM Trans. Prog. Lang. Syst., 26(2):263–300, 2004.