SuSeSim: A Fast Simulation Strategy to Find Optimal L1 Cache Configuration for Embedded Systems

Mohammad Shihabul Haque[1] Andhi Janapsatya[2] Sri Parameswaran[3]

[1] University of New South Wales, Australia
mhaque@cse.unsw.edu.au
[2] University of New South Wales, Australia
andhij@cse.unsw.edu.au
[3] University of New South Wales, Australia
sridevan@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0914
September 2009

Abstract

Simulation of an application is a popular and reliable approach to find the optimal configuration of L1 cache memory for an application specific embedded system processor. However, long simulation time is one of the main disadvantages of simulation based approaches. In this paper, we propose a new and fast simulation method, Super Set Simulator (SuSeSim). While previous methods use Top-Down searching strategy, SuSeSim utilizes a Bottom-Up search strategy along with a new elaborate data structure to reduce the search space to determine a cache hit or miss. SuSeSim can simulate hundreds of cache configurations simultaneously by reading an application's memory request trace just once. Total number of cache hits and misses are accurately recorded. Depending on different cache block sizes and benchmark applications, SuSeSim can reduce the number of tags to be checked by up to 43% compared to the existing fastest simulation approach (the CRCB algorithm). With the help of a faster search and an easy to maintain data structure, SuSeSim can be up to 94% faster in simulating memory requests compared to the CRCB algorithm.

1 INTRODUCTION

Utilizing data and instruction cache memories in computing systems improves performance and reduces energy consumption. Caches have been used to effectively reduce the ever increasing speed gap between the main memory and the processor.

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. A cache configuration is defined by different cache parameters, such as the cache size, set size or number of cache sets, associativity, and cache block size. Multiple studies [2, 5, 10, 15] have found that the correct combination of different cache parameters can reduce the energy consumption and increase the overall system performance significantly. Application specific processor design platforms such as the Tensilica's Xtensa [1, 12] 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, while a system too small will thrash, reducing performance. Thus, given an application, or a class of applications, creating a design which is optimal or near optimal for a given set of constraints will pay dividends.

Due to the non-linear nature of caches, determining cache hits and misses for a particular application can be difficult. In particular, 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 hundreds of differing cache configurations can take several months and is simply not feasible. Estimation methods (often referred to as analytical methods), depend upon heuristics and are fast but inaccurate. Strategies in [5, 7, 14, 18] are examples of estimation methods. Strategies in [4, 9, 10] use simulation of different cache configurations to produce exact estimation of total number of cache hits and misses. To speed up simulation, some of the simulation dependent approaches [9] simulate parts of the cache configurations considered. Other approaches simulate all the cache configurations under consideration extensively to maintain reliability. These are called 'Exact Approaches'. One of the widely used exact single processor cache simulation tool is Dinero IV [4], designed by Jan Elder and Mark Hill. Among the exact approaches, the CRCB algorithm [17], proposed as an enhancement to Janapsatya's method [10], is considered to be the previously fastest method.

In our research, we have analyzed exact simulation methods, especially Janapsatya's method with the proposed enhancements in the CRCB algorithm [10, 17]. We have found that these methods can be improved by clever searching methods and better data structures. We proposed a new exact cache simulation algorithm "Super Set Simulator" (SuSeSim), for L1 cache. SuSeSim overcomes most of the problems we have identified in the previously proposed simulation methods.

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 SuSeSim algorithm, and Section 5 describes the experimental setup.

2 RELATED WORK

Evaluating the performance of cache memories has been studied extensively for a long time. The approaches to evaluate caches can be broadly categorized into two: analytical; and, simulation dependent. Analytical approaches [5, 7, 14, 18] depend on heuristics, are fast to compute, but are limited in their accuracy. Simulation based approaches [4, 9, 10] usually produce error free results of cache hits and misses, and take considerably more time than analytical approaches to compute.

Several techniques are used to speed up simulation of traces to obtain cache hits and misses. The first technique is partial simulation [9], 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 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.

Because of their reliability, several attempts have been made to improve the speed of exact, concurrent, simulation based cache evaluation approaches. In 1989, Hill et al. in [8] studied the effects of associativity in caches. They introduced a forest simulation technique to 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 [6], 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. [16] proposed a binomial tree dependent cache simulation methodology to improve methods described in [8]. Sugumar's method had a time complexity of $O((log_2(X) + 1) \times A)$ for searching, where X and A are size and associativity of the cache respectively. Time complexity of maintaining the tree was $O((log_2(X)+1) \times A)$. In 2004, Li et al. [13] proposed an improvement to Sugumar's methodology by introducing a compression method to reduce simulation time.

In 2006, Janapsatya et al. [10] proposed a technique by utilizing several cache inclusion properties and a binomial tree structure. Janapsatya's top-down simulation strategy helped to speed up simulation of multiple cache configurations by reading the application trace only once. Janapsatya'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 most recent access time. Therefore, Janapsatya's method had a shorter simulation time than previously proposed solutions. Janapsatya's method had a fixed time complexity of $O((log_2(X)+1) \times A)$ 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 data structure was $O(log_2(X) + 1)$. In 2009, Tojo et al. [17] proposed two enhancements to Janapsatya's method. These pruning based proposals made the simulation even faster by reducing the number of addresses to be examined. These approaches are known as the CRCB algorithm. Due to the importance in our research, Janapsatya's approach with the CRCB enhancement has been described in the following section.

3 BACKGROUND

Cache configurations are 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 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, $T = S \times B \times A$.

We have presented an example of a set associated cache in Figure 3.1. The cache has eight data storage locations. The amount of data that can be stored in each data storage is called block size (B). Every two data storage locations form a set in Figure 3.1. Therefore, we have four sets ($S = 4$), and each set has two different locations to store data ($A = 2$) and each of these locations are called cache ways. Therefore, the cache of Figure 3.1 is a two-way set associative cache. Each set is identified by an index number. In Figure 3.1, $T = 8$ bytes for $B = 1$ byte.

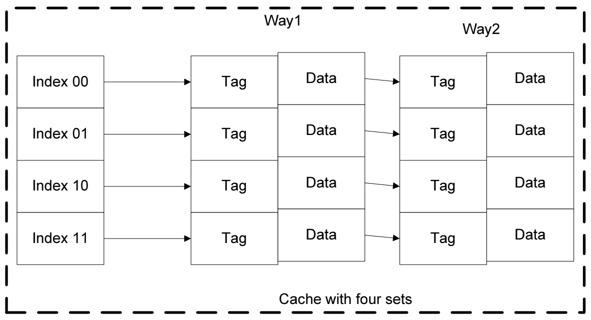


Figure 3.1: A set associative cache

In Figure 3.2, we have shown how a byte addressable memory address content is searched in the cache of Figure 3.1. Let's consider that the cache has block size of two bytes ($B = 2$). To search the content from byte addressable memory address shown in Figure 3.2, the last one bit is used to select the byte inside the two byte cache block. Therefore, the last bit of the address is called the byte offset. As $S = 4$ in the cache of Figure 3.1, the penultimate two bits of the address of Figure 3.2 are used to select the cache set. The rest of the address is used as tag. If the tag is found in the cache of Figure 3.1 inside any way of index 00, it will be a hit; otherwise, it will be a miss. On a miss, content from the memory address of Figure 3.2 will be placed inside the cache set 00 of Figure 3.1.

It has been found by the previous researchers [10, 17, 8, 16] that the LRU replacement policy helps to make the simulation process faster. LRU replacement policy enables the simulator to use the following two observations to speed up simulation by reducing total number of cache sets to be simulated:

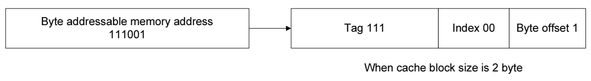

Byte addressable memory address
111001
Tag 111
Index 00
Byte offset 1
When cache block size is 2 byte

Figure 3.2: A byte addressable memory address

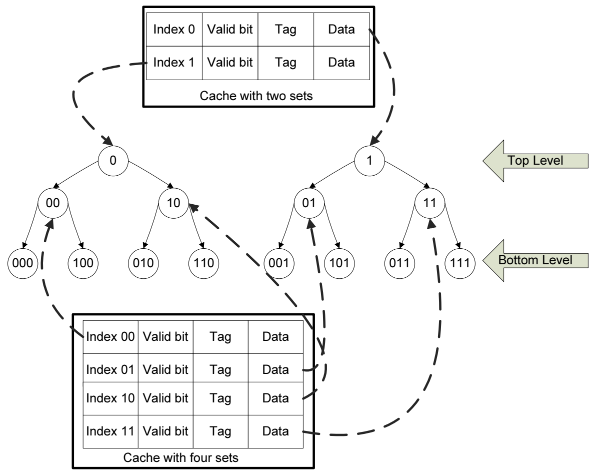


Figure 3.3: Formation of simulation tree

1. Given caches with the same associativity and block size, and using the Least Recently Used (LRU) replacement policy, whenever a cache hit occurs, all caches that have larger set sizes will also guarantee a cache hit; and,
2. Hit on a set associative cache means a cache hit is guaranteed on all the caches with larger associativity and same block size.

The above observations are described in detail in [10].

To utilize the above mentioned observations, special data structures are created. A set of trees, called "Simulation trees", are created such that each node reflects a set in a cache and each level of a tree represents a cache configuration. Inside a simulation tree, all the cache configurations must have the same cache block size. An example has been presented in Figure 3.3. In Figure 3.3, the two nodes at the top of the two trees point to a cache with $S = 2$. The first node on the left, marked '0', refers to the cache set with index 0. And the second node, marked '1', refers to the 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 3.3. Similarly, the third level (or depicted as bottom level in Figure 3.3), will represent a cache with eight sets. For caches with larger set numbers, the tree is further expanded with greater number of levels.

The advantage of such a structure in cache 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). 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. Moreover, only one tree is needed during the simulation of an address. Due to this strategic tree structure, a large number of caches with differing sizes can be simulated simultaneously, with minimum number of searching. Note that the tag field in the cache with $S = 2$ will be filled with the number '1000101' and in the cache with $S = 4$ the tag field will be filled with the number '100010'.

To store address tags, in [10, 17], the authors associated a singly linked list with each simulation tree node. Each linked list node corresponds to a cache way. Figure 3.4 presents an example of such a linked list. Four nodes inside the example linked list in Figure 3.4 indicate that a four way set associative cache is under simulation. Only the most recently used memory address tags are stored in these linked list nodes. So, in the example list, the tag inside the first way (also called the head node) is the most recently accessed tag, the tag inside the second way is the second most recently accessed one and so on. Let's describe the advantage of this type of linked list with the aid of an example. Let's consider that at a certain point in time, the processor requests the last four memory addresses presented in Table 3.1. If the cache has one byte cache block and memory is byte addressable, all except the first requested addresses shown in Table 3.1 will go to index 0 of the two-set cache of Figure 3.3. At the end of the last request, index 0 of the two-set cache will look like the cache set shown in Figure 3.9. It can be seen that in Figure 3.9, the most recently accessed tag "11000" is directly accessible from the tree node; however, to access tag "11100", the node with tag "11000" must be searched first. Similarly, to access tag "00110", the nodes with tag "11000" and "11100" must be searched first. Therefore, the node arrangement policy in the linked list helps to search recently accessed tags quickly. As temporal locality increases the chance of reusing recently accessed tags in the near future, the node arrangement policy in the linked list in [10, 17] helps to reduce simulation time.

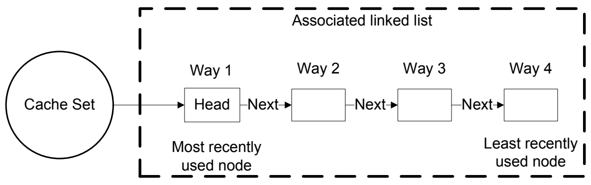


Figure 3.4: Singly linked list to represent associativity

Figure 3.5 gives an example of how memory addresses are divided into different parts and used to select cache sets when we have the simulation tree of Figure 3.3. In this example, our memory is byte addressable and our cache block size is 2 bytes; and requested binary memory address is 111101. From Figure 3.5 the values of tag, index and offset for the example address can be seen, in the three different cache configurations under simulation. Figure 3.6 shows the tags of tree 1 of Figure 3.3 in the associated linked list when processor requests the binary addresses from Table 3.1 simultaneously.

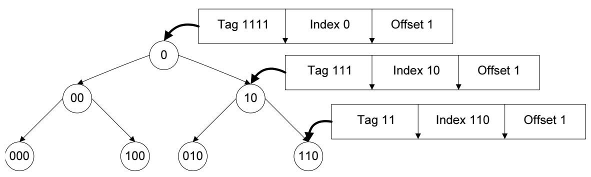


Figure 3.5: Values for tag, index and offset for a requested address in different cache configurations

Application
111101
101100
011000
001100
111000
110000

Table 3.1: Trace of requested addresses

In the following subsection, we are going to show, how these simulation trees are used in Janapsatya's algorithm with the CRCB enhancements [10, 17] to perform a fast simulation.

3.1 Janapsatya's methodology with proposed enhancements in the CRCB algorithms

To record the total number of cache misses for different cache configurations, Janapsatya's approach [10] keeps a table, indexed with 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. An example miss counter table has been presented in Figure 3.7. In this example table, the total number of misses of the first entry will be increased only if a tag is missed in the cache with set size 1, block size 1 byte and associativity 1. Similarly, the second entry's total number of misses will be increased when we want to record a miss for a cache with set size 1, block size 1 byte and associativity 2. This example miss counter table can hold total number of misses for caches with set size 1 to 1024, associativity 1 to 512 and block size 1 to 512 bytes.

Janapsatya's simulation method has three phases: Tree formation, Tag searching and Cache set update. In the following subsections the phases are described.

Tree formation

At the beginning of simulation in Janapsatya's approach, a forest of simulation trees (described above) is created for each cache block size. 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 using the procedure we have described earlier in the beginning of Section 3.

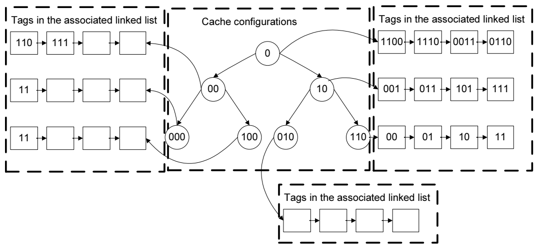
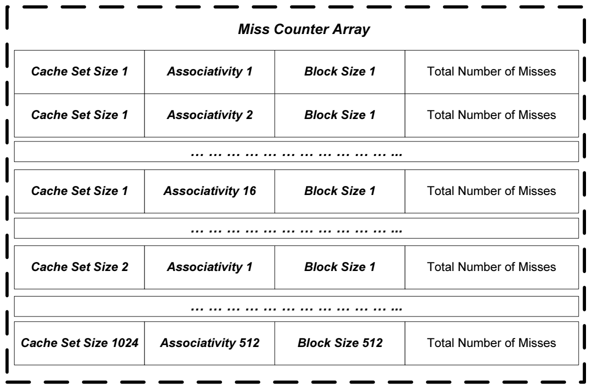

Tags in the associated linked list
110
111
11
11
Cache configurations
0
00
10
000
100
010
110
Tags in the associated linked list
1100
1110
0011
0110
001
011
101
111
00
01
10
11
Tags in the associated linked list

Figure 3.6: Address tags in different cache configurations



Miss Counter Array			
Cache Set Size 1	*Associativity 1*	*Block Size 1*	Total Number of Misses
Cache Set Size 1	*Associativity 2*	*Block Size 1*	Total Number of Misses
... ...			
Cache Set Size 1	*Associativity 16*	*Block Size 1*	Total Number of Misses
... ...			
Cache Set Size 2	*Associativity 1*	*Block Size 1*	Total Number of Misses
... ...			
Cache Set Size 1024	*Associativity 512*	*Block Size 512*	Total Number of Misses

Figure 3.7: Miss counter array of Janapsatya's method

Tag searching

To simulate each address, Janapsatya's approach start from the top level of a simulation tree or smallest cache configuration and continues toward the biggest configuration. Each tree level is simulated in sequence. Inside each level of the tree, the tag is searched in the associated singly linked list of the selected tree node (cache set). Tag searching starts from the head node inside the cache set and continues toward the node with least recently used tag or tail. The search continues until a tag is found or there are no more tags left to check in the linked list. In Figure 3.8, direction of simulation and direction of tag searching are presented.

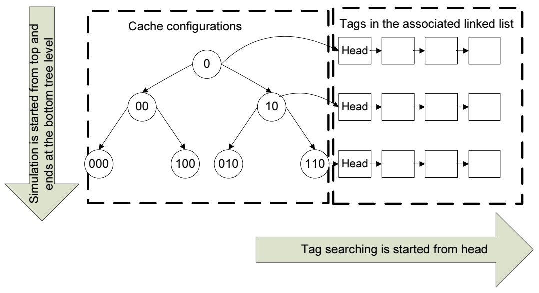

Simulation is started from top and ends at the bottom tree level
Cache configurations
0
00
10
000
100
010
110
Tags in the associated linked list
Head
Head
Head
Tag searching is started from head

Figure 3.8: Direction of simulation and tag searching

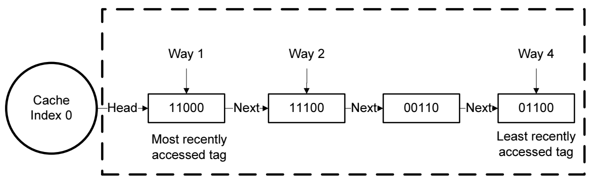


Figure 3.9: Singly linked list with requested address tags

Cache set update

Depending on the outcome of the tag search inside a cache set, one of the following actions can be taken:

1. If the tag is missed, the miss counters for all the cache configurations with set size equal to the current configuration, and associativity less than or equal to the current configuration's associativity will be increased. This is because, according to the second observation of Section 3, when all the caches are using LRU replacement policy, a miss on a set associative cache guarantees miss on caches with smaller associativity and equal set size. On a miss, Janapsatya's simulator goes through five different steps to place the missed tag in the linked list. In Figure 3.10, all of these steps are shown. In step 1 and 2 the entire linked list must be searched to point out the least recently used node, or tail, and second least recently used node, or $(tail - 1)$. Next, the missing tag will be placed in the tail of the linked list and in the next step, tail will be moved to the head of the linked list. In step 5, $(tail - 1)$ will become the new tail. We would like to mention that depending on implementation, some steps of Figure 3.10 may be combined; however, tasks inside these steps must be performed to update the missed tag. Here we show five steps to increase readability. The missed tag update process can be explain with the help of an example. Let's suppose at a certain point in time, index 1 of a two set cache looks like Figure 3.11 (assuming byte addressable memory and one byte cache block). The processor requests address 011111 which is not available in index 1. To place the missed address tag, least recently accessed tag 11111 will be replaced by 01111. Tag 01111 will become the most recently accessed address tag and second least recently accessed address tag 11110 will become the new least recently accessed address tag or tail. After all this modification, cache index 1 of Figure 3.11 will look like Figure 3.12

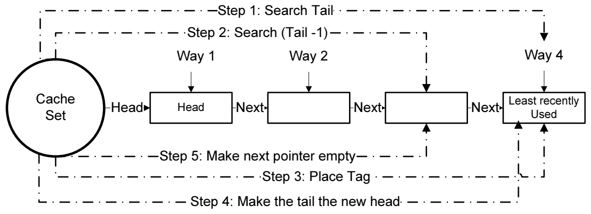


Figure 3.10: Janapsatya's method needs five steps to update address tag inside the associated singly linked list

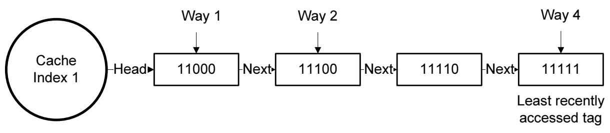


Figure 3.11: Index 1 of a two-set cache with some tags

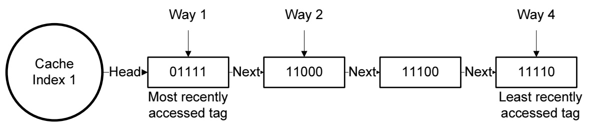


Figure 3.12: Index 1 of the two-set cache of Figure 3.11 after the missed tag is update

2. If the searched address tag is found at the linked list node in the V^{th} way, the node will be moved to the head of the linked list. However, to do that, the simulator goes through three different stages. First of all, the node before the V^{th} way/node (the $(V-1)^{th}$ way/node) must be searched out. The next pointer of the $(V-1)^{th}$ node will be set to point to the next element of the V^{th} node. After that, the V^{th} node will become the new head. Miss counter for all the cache configurations with set size equal to the current set size and associativity less than V will be increased. We can use the same example of Figure 3.11 to explain the scenario. Let's suppose, address 11110 is requested. It is available at the third way of cache index 1. So, tag 11110 will be sent to the head, and tag 11111 will become the next tag of tag 11100. After updating the found tag position, cache index 1 of Figure 3.11 will look like Figure 3.13.

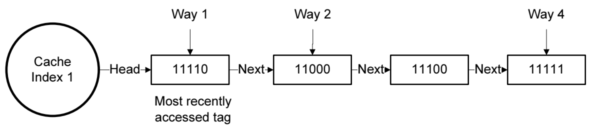


Figure 3.13: Index 1 of the two-set cache of Figure 3.11 after the hitted tag is update

The CRCB algorithm

In 2009, Tojo et al. proposed the following two enhancements for Janapsatya's method [10] in their CRCB algorithm [17]:

1. The most recently accessed tag of a cache set must not be searched in the bigger set sized cache configurations as they will definitely be found in the heads of those configurations. This enhancement helps to use the first observation presented in Section 3 effectively to reduce total number of caches to be simulated. The first observation presented in Section 3 says, "Given two caches with the same associativity and block size, and using the Least Recently Used (LRU) replacement policy, whenever a cache hit

occurs, all caches that have larger set sizes will also guarantee a cache hit". However, the observation doesn't give any hint where the hit will be inside the cache set. Therefore, without the enhancement of CRCB, to update miss counters for caches with different associativities, each and every level of the selected simulation tree must be simulated.

2. If the same memory addresses are requested consecutively, no need to simulate the later requests except the first one as the duplicated requests are always going to generate hits in the head node of all the cache configurations in a forest.

3.2 Some additional observations

In this section, we are going to describe two additional observations that help us to simulate cache configurations even faster. The observations are as follows:

1. According to the first observation described in Section 3, any tag available inside a cache must be available in a bigger cache with the same associativity and cache block size, when all the caches use the LRU replacement policy. So, the bigger cache is a super set of the smaller cache. This fact was discussed in [10]. From the first observation described in Section 3, we can also say that a tag that is not present in the bigger cache has no possibility of being in the smaller cache.
2. Our second observation is, provided that all the caches use the LRU replacement policy and all of them have same cache block size and associativity, a tag found in a cache way of a cache set cannot be available inside a more recently used cache way inside a smaller cache.

Assume that caches *C1* and *C2* have the same associativity A and block size B, but *C2* is double the size of *C1*. If we are using the same application trace to simulate both *C1* and *C2* at the same time, at a certain point in time the same address must be requested from both caches. Inside each set of *C1* and *C2*, tags are arranged according to their last access time due to LRU replacement policy. Therefore, each cache set inside *C1* and *C2* is an ordered set. Due to the simulation tree structure, at a particular point in time, tags available in a set of *C1* will be available in two different sets of *C2*. An example of a simulation tree has been presented in Figure 3.14. In this Figure, two caches have been presented. The cache at the level 1 of the tree has only one set and associativity is four. The other cache, presented in the level 2, also has an associativity of four; however, it has two cache sets, Set 0 and 1. Both of the caches have same block size. If these two caches are simulated at the same time with the same application trace, tags that go to the set of the cache in the tree level 1, must go to either set 0 or 1 of the cache presented in the tree level 2.

Let's suppose tags of the cache set S of *C1* are available in two different cache sets *S1* and *S2* of *C2*.

As, $S \subset (S1 \cup S2)$ and both of the caches are using LRU replacement policy, the tags of S are the most recently accessed A tags of $(S1 \cup S2)$. For *S1* and *S2*, one of the following cases is possible:

1. Either *S1* or *S2* is an exact copy of *S*. Therefore, the M^{th} most recently accessed tag of $S1$ or $S2$ will be the M^{th} most recently accessed tag of S.

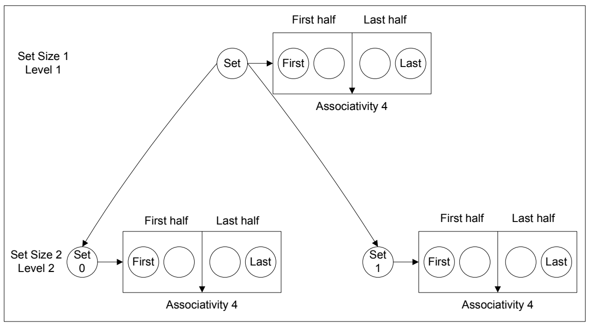
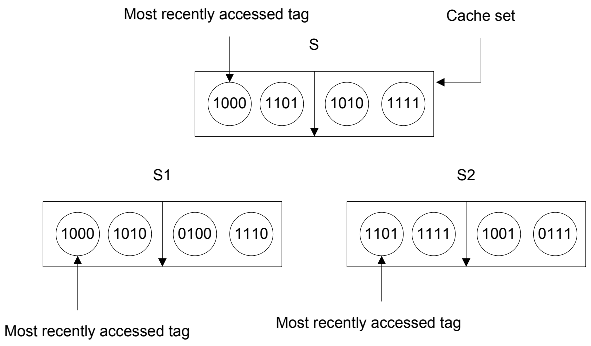


Figure 3.14: simulation tree for two cache configurations

2. In either *S1* or *S2* (for this case let us assume *S1*) the most recently accessed M tags, when $M < A$, are identical to the most recently accessed M tags of S, and the least recently accessed $(A - M)$ tags of S are available in the other set (in this case *S2*), and those $(A - M)$ tags will be in the most recently used positions of *S2*.

These two cases show that due to the LRU replacement policy, whatever is found in *S1* or *S2* at the X^{th} $(X = 1, 2, 3, ..., A)$ most recently accessed position, if available in S, it must be available at one of the positions among X^{th} to the A^{th} most recently accessed position (i.e. least recently accessed position) in S.

The following example illustrates this point. Let's suppose at a particular point in time, S, *S1* and *S2* have the tags for binary memory addresses presented in the circles in Figure 3.15. In *S2*, the tag for binary memory address 1101 is the most recently accessed tag, however, for S, it is the second most recently accessed tag. Again, in *S1*, the tag of 1010 is the second most recently accessed tag; however, in S, it is the third most recently accessed tag.



Most recently accessed tag
Cache set
S
1000
1101
1010
1111
S1
1000
1010
0100
1110
S2
1101
1111
1001
0111
Most recently accessed tag
Most recently accessed tag

Figure 3.15: Cache sets S, S1 and S2 with associativity four

If we know that the requested tag is the X^{th} most recently accessed tag in *S1* or *S2*, where $X > (A/2)$, according to our second observation, the tag is not going to be one of the most recently accessed $(X - 1)$ tags in S. If we search from the most recently accessed tag in S to search the X^{th} most recently accessed tag of *S1* or *S2*, if it is available in S at the X^{th} most recently accessed cache way, X tags will be searched inside S. However, if we searched from the least recently accessed tag, only $(A - X + 1)$ tags are needed to be searched inside S. As $X > (A/2)$, $(A - X + 1) < X$. One example has been given in Figure 3.16 using the caches presented in Figure 3.15. Note that in Figure 3.15, binary memory addresses are shown in the circles, and in Figure 3.16, address tags of the binary addresses of Figure 3.15 have been shown in the circles. Figure 3.16 shows that after finding the tag 110 in the bottom tree level, if the tag is available in the top tree level, it must be available within the last two nodes in a cache set. 110 is the address tag for binary memory address 1100 when cache block size is one byte. If searched from the most recently used tag in the top tree level, three nodes are needed to be searched to search the tag for 1100. However, if the tag is searched from tail to head, only two nodes need checking. If the top tree level doesn't have the tag, head to tail search searches the entire cache. On the other hand, just by performing the tail to head search among the last two nodes in the smaller cache, simulator can determine whether the tag is there or not. Therefore, addition of new search function that searches from tail to head can reduce number of searches. Therefore, we can conclude that when the tag is at the X^{th} way of a cache set in the bigger cache where $X > (A/2)$, depending on the value of X, tail to head searching can reduce the number of ways searched inside a set of the smaller cache from one to $(A/2)$. In other words, the tail to head searching can reduce searching up to 50% compared to the head to tail search.

To benefit from our observation of Section 3.2, we need to start searching from bottom level of the tree or biggest cache configuration and continue towards the smallest cache configuration. In other words, we need bottom-up simulation inside the tree. In addition, bottom-up simulation combines the first observation of Section 3 and the first observation of the CRCB enhancement by using our first observation. If the tag is found in the head node of a cache set during bottom-up simulation, just skip to the next smaller cache configuration (without any update). On a miss, there is no need to search the tag in the upper levels of the simulation tree when a bottom-up simulation is utilized.

Deployment of these two additional observations requires following two modifications to the data structure to represent associativity inside a simulation tree for faster performance.

1. To have a reverse tag search function (tail to head) and a head to tail tag search function requires a doubly linked list instead of singly linked list to be associated with a simulation tree node. Therefore, each linked list node not only points to its next element but also to its previous one. One example of such a doubly linked list has been presented in Figure 4.1. In addition, a doubly linked list allows simple updating of a tag position inside a cache set without searching or remembering the previous node inside a linked list.
2. On a miss, there is no need to search the tag in the smaller cache configurations; however, tag must be updated in all of those caches. To make the update process easy and fast, tree nodes needed to be connected to the tail node as well as the head node of the associated linked list. On a miss, the update process will go to the tail of a cache set, place the tag and make the tail the new head of the tree node.

Based upon the observations and the new data structure, we have proposed our new L1 cache simulation algorithm 'SuSeSim' for different cache configura-

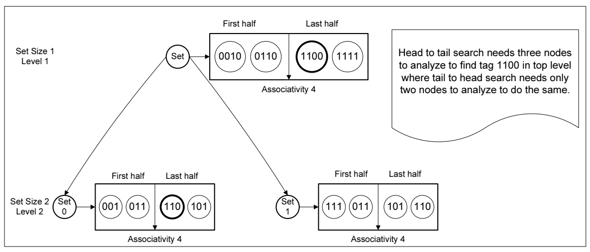

First half
Last half
Set Size 1
Level 1
Set
0010
0110
1100
1111
Associativity 4
Head to tail search needs three nodes to analyze to find tag 1100 in top level where tail to head search needs only two nodes to analyze to do the same.
Set Size 2
Level 2
Set 0
001
011
110
101
Set 1
111
011
101
110
Simulation tree for two cache configurations

Figure 3.16: Searching non frequent address tag

tions with the same block size in the following section.

4 SuSeSim Algorithm

SuSeSim simulates caches with the LRU replacement policy. Similar to Janapsatya's method, SuSeSim has three phases: Tree formation, Tag searching and Cache set update. In the following subsections, we describe these three phases in detail.

4.1 Tree formation

A cache miss counter table similar to the one presented in Figure 3.7, and a simulation tree like the one presented in Figure 3.3 is used in SuSeSim. However, the simulation tree nodes have doubly linked lists instead of singly linked lists, to simulate set associative caches. Nodes in the doubly linked lists correspond to cache ways. In addition, each tree node not only points to the most recently used address tag (head of the associated doubly linked list), but also points to the least recently used address tag (tail of that list).

At the beginning of simulation, a forest of simulation trees (described in Section 3) is created for each cache block size. Processor requested addresses are read from the selected application's trace file one at a time and sent to each forest. A duplicated address is not send to a forest when requested consecutively. Inside a forest, a tree is selected using the procedure we have described in Section 3.

4.2 Tag searching

To simulate a memory request, SuSeSim starts searching the address tag in the biggest cache configuration and continues toward the smallest cache configuration. Each tree level is simulated in sequence. Inside each level of the tree, tag is searched in the associated doubly linked list of the selected tree node (cache set).

In SuSeSim, there are two different search functions to search a tag inside the doubly linked list associated with a simulation tree node.

1. A search function that searches from the head to tail in the cache set. It is the default search function.

2. The second search function starts searching the address tag from the least recently accessed tag or tail and continues searching in the previous N nodes where N is the given number of nodes. This search is performed instead of default search in the parent tree level, if an address tag is found as the X^{th} most recently accessed tag in one node of the current simulation tree level, where $X > (A/2)$ and A is associativity. The value of N will be $(A - X + 1)$ during the simulation of the parent tree level. We call this search the reverse search.

A search function will be stopped when the tag is found. Figure 4.1 shows the search paths of these two search functions.

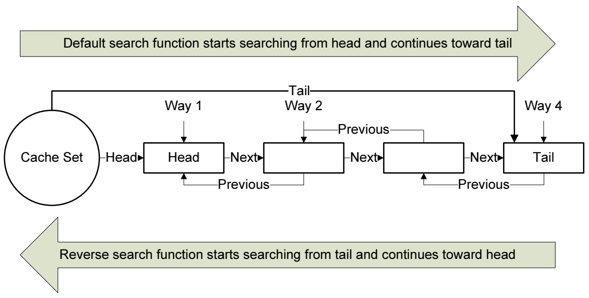

Default search function starts searching from head and continues toward tail
Tail
Way 1
Way 2
Way 4
Previous
Cache Set
Head
Head
Next
Next
Next
Tail
Previous
Previous
Reverse search function starts searching from tail and continues toward head

Figure 4.1: Doubly linked list and search paths of SuSeSim algorithm

4.3 Cache set update

Depending on the outcome of the search, one of the following actions is taken.

1. On a miss, there is no need to search the address tag in the smaller cache configurations. The missing address tag will be placed in the tail node of the appropriate cache sets of the current and other smaller cache configurations in the simulation tree. After that, the tail will be moved to the head node of the linked list. The node with the least recently accessed address tag will become the new tail. Cache miss counters for all of the configurations with any associativity and set size equal to these updated cache configurations will be increased.
2. If the tag is found in the head of the associated doubly linked list, simulation is skipped from the current configuration to the next smaller cache configuration in the simulation tree.
3. If the tag is found in the V^{th} node/cache way in a cache set when the V^{th} node/cache way is neither the head nor the tail, the miss counter for all the cache configurations with set size equal to the current set size and associativity less than V will be increased. The tag container node of the linked list will be brought to the head position of the linked list and the node with the least recently accessed address tag will become the new tail. After that, the address tag will be searched in the next smaller cache configuration; however, if $V > (A/2)$, the reverse search function will be used. The reverse search function will search up to the V^{th} node if the tag is not found.

Three examples are given to illustrate the three different actions of simulation process. Let's suppose that we have the simulation tree shown in Figure 4.2. Here, we have two set associative caches to simulate. Both these caches have associativity of four. We want to simulate the following three cases:

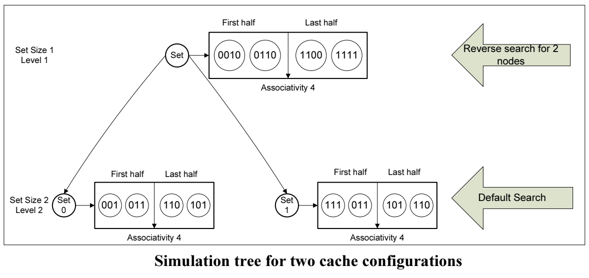


Figure 4.2: An example simulation tree

1. We want to simulate the binary byte addressable memory address 1100 when cache block size is one byte. SuSeSim will start simulation from the bottom level of the tree. The default search function will be used at the beginning. We can see from the Figure 4.2 that the tag for memory address 1100 is in set 0. This tag is the third most recently accessed tag of the cache set. After finding the tag, SuSeSim will increase the cache miss counter for each of those configurations that has set size two, associativity less than three and cache block size one byte. After that, Set 0 will look like Figure 4.3 as the tag for 1100 will become the new head due to the LRU replacement policy. After finishing simulation in the bottom level, SuSeSim will search the tag in the top tree level; however, the reverse search function will be used, as in the bottom level, the tag was found as the third most recently accessed tag. Reverse search will search the tag in the two least recently accessed tags in the top level. This time the tag will be found as the third most recently accessed tag again; therefore, the miss counter for all the cache configurations with set size one, associativity less than or equal to three and cache block size one byte will be increased. After that, the tag for address 1100 will become the head of this cache set again.

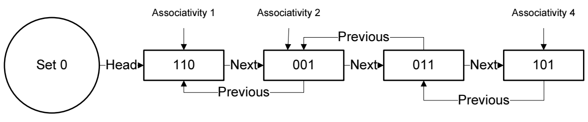


Figure 4.3: Cache set 0 of Figure 4.2 after update of tag position

2. If we want to search for the tag corresponding to address 0010, it will be found at the head tag in the bottom tree level. Therefore, no miss counter update and linked list update will be performed. The default search will be performed again in the top tree level. As the tag is again the most recently accessed tag in the top level, no change will occur.
3. If we want to simulate address 1000, default search in the cache set 0 of the bottom tree level will fail to find the tag which makes SuSeSim aware

that the tag is not available in the search space. Therefore, the miss counter of all the cache configurations with set size equal or less than two, associativity equal or less than four and cache block size one byte will be increased. The tag for 1000 will be added as the head node in both set 0 of bottom tree level and cache set of the top tree level. Figure 4.4 shows the tree after all these modifications.

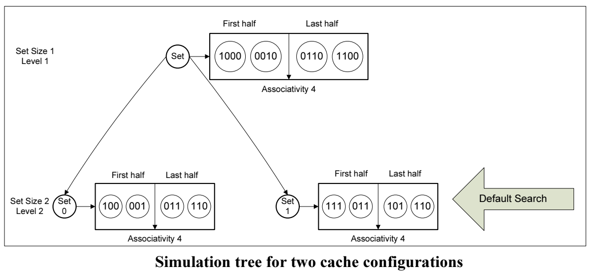


Figure 4.4: Search tree of Figure 4.2 after the placement of new tag

It should be mentioned that, like Janapsatya's approach with the CRCB algorithms, in SuSeSim, each level of a simulation tree except the top level must be twice as big as its parent level's cache.

The SuSeSim algorithm has been presented in Algorithm 1.

5 EXPERIMENTAL SETUP

With the implementation described above, SuSeSim can reduce total simulation time. Each SuSeSim doubly linked list entry is used to hold a tag (32 bits) and pointers to the next and previous elements (32 bits each). In total, each doubly linked list entry needs to store 96 bits. In the simulation tree, each node keeps a pointer to the head element (32bits) and tail element (32 bits) of the linked list; giving a total of 64 bits. Therefore, per tree node or cache set is $(64 + (96 \times A))$ bits, where A is the maximum associativity. Janapsatya's method (and CRCB) needs $(96 + (65 \times A))$ bits per tree node or cache set.

We have re-implemented Janapsaty's method with and without the CRCB enhancement. All of these simulators are written in C++. We have compiled and simulated programs from Mediabench [11] with SimpleScalar/PISA 3.0d [3]. Program traces were generated by SimpleScalar and fed into all of these three methods. We have verified hit and miss numbers of each method using DineroIV [4] and found all of them are consistent with each other. Simulations were performed on a machine with dual core Opteron64 2GHz processor and 8GBytes of main memory.

We have simulated 1300 different cache configurations (for both data and instruction cache) with each simulator. The cache configurations are the all possible combinations of the cache parameters presented in Table 5.1.

In an embedded system, cache block sizes (B) of 128, 256 and 512 bytes are not practical choices. However, we have studied those large cache block sizes to check the cases where the cache miss rates are very low (and to compare with the CRCB system).

Algorithm 1 SuSeSim Algorithm

```
1: while Trace is not finished do
2:     Read an address request Addr;
3:     offset=maximum cache set size;
4:     found=true;
5:     previous_result=0;
6:     while offset is not empty do
7:         tag_s=Tag from Addr for offset;
8:         index_s=cache index from Addr for offset;
9:         Select tree with cache set index_s and go to level for cache set size equal to
           offset;
10:        if found is false then
11:            Place tag in the tail of the associated doubly linked list.;
12:            Move tail to be the head of the linked list;
13:            Increase cache miss counter for all cache configurations with set size equal
               to offset and any associativity;
14:        else
15:            Goto the doubly linked list associated with tree node with index index_s;
16:            if head of tree node with index index_s does not contain tag_s then
17:                if previous_result>(maximumassociativity/2) then
18:                    Search the doubly linked list to find a tag entry equal to tag_s, within
                       tail to previous_result^{th} node;
19:                else
20:                    Search the linked list to find a tag entry equal to tag_s within head to
                       tail;
21:                end if
22:                if a cache hit occurs in the $S^{th}$ element of the linked list then
23:                    Increase cache miss counter for all caches with set size equal to offset
                       and associativity less than S.;
24:                    Make the $(S-1)^{th}$ node the new tail of the doubly linked list when $S^{th}$
                       node is the tail node;
25:                    Move the $S^{th}$ node to be the head of the doubly linked list;
26:                    found=true;
27:                    previous_result=S;
28:                else
29:                    found=false;
30:                    Place tag in the tail of the linked list;
31:                    Move tail to be the head of the linked list and make the (tail-1) node
                       the new tail of the doubly linked list;
32:                    Increase cache miss counter for all cache configurations with set size
                       equal to offset and any associativity;
33:                end if
34:            end if
35:        end if
36:        offset=offset/2;
37:    end while
38: end while
```

Six Mediabench applications were used to verify the system. These are: G721 encode, G721 decode, JPEG encode, JPEG decode, MPEG2 encode and MPEG2 decode. The numbers of memory address requests have been presented in Table 5.2 for each of the used applications. For each application, we have

Cache Set Size=2^i	where $0 <= i <= 12$
Cache Block Size=2^i Bytes	where $0 <= i <= 9$
Associativity=2^i	where $0 <= i <= 9$

Table 5.1: Cache configuration parameters

Application	Number of requests
Jpeg encode(CJPEG)	25680911
Jpeg decode(DJPEG)	7617458
G721 encode	154999563
G721 decode	154856346
Mpeg2 encode	3738851450
Mpeg2 decode	1411434040

Table 5.2: Trace files used for simulation

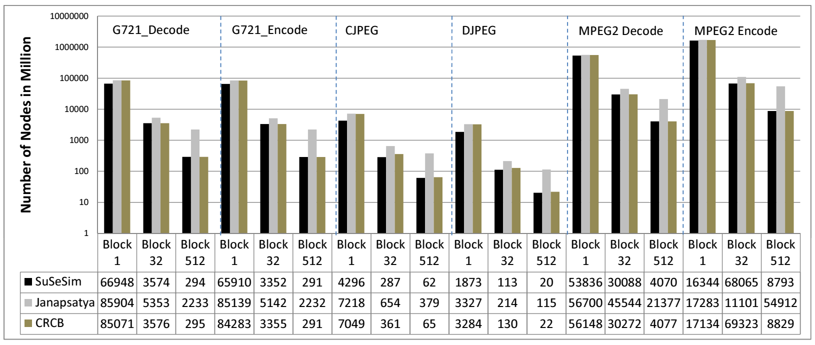
recorded the following:

1. Total number of doubly linked list nodes, associated with tree nodes, searched for requested address tags during the simulation of the entire application trace.
2. Total number of cases where the requested address tags were found in the head node of a tree node's doubly linked list.
3. Total time elapsed for searching tags inside cache sets.
4. Total time elapsed for the simulation.

These results are shown in Figures 5.1, 5.2, 5.3 and 5.4 respectively. In these Figures, results are presented for cache block sizes of 1 (Lowest), 32 (Middle) and 512 (Highest).

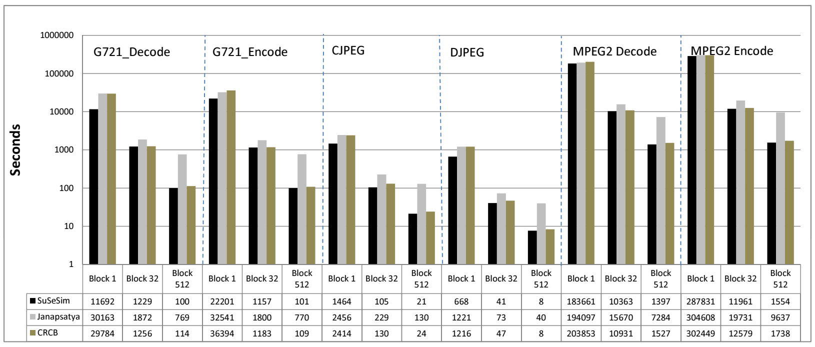
From the simulation results it can be seen that SuSeSim is the fastest simulator compared to Janapsatya's approach and the CRCB algorithm. It reduces simulation time via the following two methods:

1. From Figure 5.1, it can be seen that SuSeSim searches the fewest number of linked list nodes during simulation. Searching simulation trees from bottom to top and using two different searching strategies are the reasons behind it. Bottom-up searching searches the fewest nodes when a tag is missing in the entire search space. On the other hand, Janapsatya's approach and CRCB perform their highest number of searches when a tag is missed as each cache set needs to be searched from the start to the end. Besides that, reverse search reduces number of searches for less frequently used address tags. As less number of nodes are needed to be searched, simulation time automatically decreases. Note that for both SuSeSim and the CRCB algorithm, those searches that find tags in the head nodes have been ignored in our count. This is because, both SuSeSim and CRCB algorithm do nothing in such a scenario. However, Janapsatya's original plan continues simulation in the bigger configurations even after finding a tag in the head node. On average, SuSeSim finds 25% of tags in the head node where the CRCB algorithm finds 4% of tags in the head, among the simulated nodes. As less number of searches are performed in SuSeSim



	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512
■ SuSeSim	66948	3574	294	65910	3352	291	4296	287	62	1873	113	20	53836	30088	4070	16344	68065	8793
■ Janapsatya	85904	5353	2233	85139	5142	2232	7218	654	379	3327	214	115	56700	45544	21377	17283	11101	54912
■ CRCB	85071	3576	295	84283	3355	291	7049	361	65	3284	130	22	56148	30272	4077	17134	69323	8829

Figure 5.1: Total number of linked list nodes searched during simulation



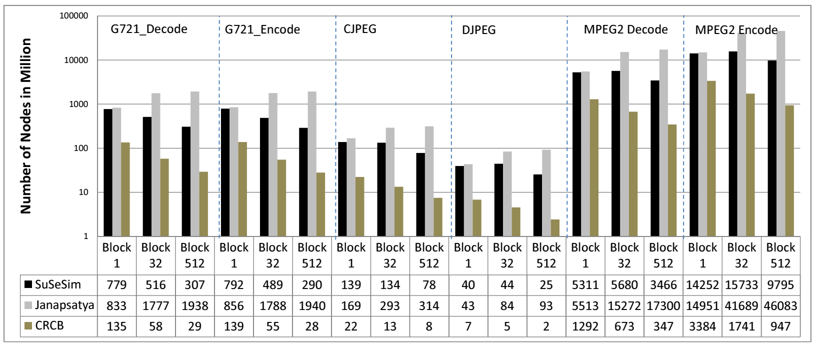
	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512
■ SuSeSim	11692	1229	100	22201	1157	101	1464	105	21	668	41	8	183661	10363	1397	287831	11961	1554
■ Janapsatya	30163	1872	769	32541	1800	770	2456	229	130	1221	73	40	194097	15670	7284	304608	19731	9637
■ CRCB	29784	1256	114	36394	1183	109	2414	130	24	1216	47	8	203853	10931	1527	302449	12579	1738

Figure 5.2: Total time elapsed for searching tags during simulation

and almost one-fourth of those searches find a tag in the head, the number of effective searches and the simulation time decreases. Figure 5.2 shows total time needed to search for tags for different applications. Figure 5.3 shows total number of nodes, among the simulated linked list nodes, found in the head node for different applications in different simulators.

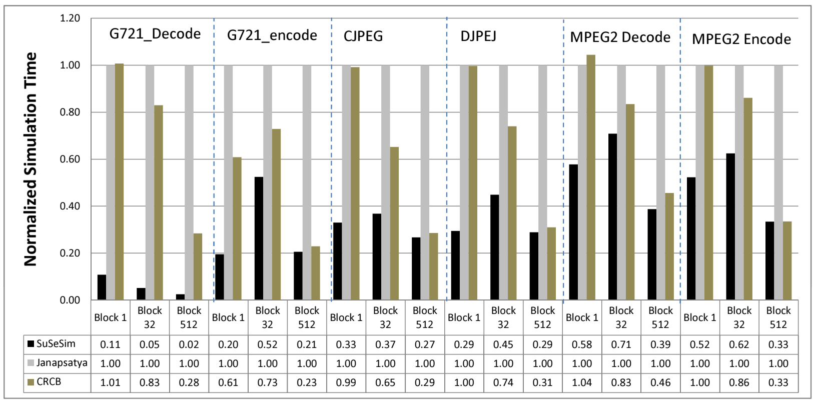
2. The second reason for faster simulation is having connection of the tail node of the associated linked list with the tree node. Since SuSeSim can determine the absence of a tag without searching inside a cache set, connection between the tail and the tree node makes the missed tag update less time consuming. Without the connection between tail and tree node, tail must be found first (by traversing the entire list) to place a missed tag. Searching the tail is a time consuming process as all the nodes of the associated linked list must be searched.

Depending upon different cache block sizes and benchmark applications, SuSeSim can reduce the number of tags to be checked up to 43% (excluding tags in the head node) compared to the CRCB algorithm, and up to 87% compared to Janapsatya's method (see Figure 5.1). Similarly, SuSeSim can reduce the time of tag searching from 2% to 61% compared to the CRCB algorithm, and 6% to 87% compared to Janapsatya's method (see Figure 5.2). In Figure 5.4, we have presented a graph to show simulation time of different simulators normalized with respected to Janapsatya's method. The graph shows that SuSeSim can run



	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512
■ SuSeSim	779	516	307	792	489	290	139	134	78	40	44	25	5311	5680	3466	14252	15733	9795
■ Janapsatya	833	1777	1938	856	1788	1940	169	293	314	43	84	93	5513	15272	17300	14951	41689	46083
■ CRCB	135	58	29	139	55	28	22	13	8	7	5	2	1292	673	347	3384	1741	947

Figure 5.3: Total number of address tags found in the head of the linked list



	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512	Block 1	Block 32	Block 512
■ SuSeSim	0.11	0.05	0.02	0.20	0.52	0.21	0.33	0.37	0.27	0.29	0.45	0.29	0.58	0.71	0.39	0.52	0.62	0.33
■ Janapsatya	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
■ CRCB	1.01	0.83	0.28	0.61	0.73	0.23	0.99	0.65	0.29	1.00	0.74	0.31	1.04	0.83	0.46	1.00	0.86	0.33

Figure 5.4: Total simulation time

up to 94% faster than CRCB and almost 98% faster than Janapsatya's method.

When a tag is not available in the simulation tree, time complexity of SuSeSim tag searching is only $O(A)$, when A is the maximum associativity a cache can have in the simulation tree, as only one linked list node in the biggest configuration needs to be searched. Address tags that are pruned due to common consecutive requests are excluded from the complexity calculations. If the tag is available in all the configurations but not in the head list, time complexity for simulation is $O((log_2(X)+1) \times A)$ at best, where X is the maximum cache set size in the search space. For other cases time complexity for hit/miss determination varies in between $O(A)$ and $O((log_2(X)+1) \times A)$ when pruned requests are excluded. Tag cannot be found in the head list of all configurations as SuSeSim does not simulate the same address requests one after another. In case of the CRCB algorithm, when a tag is absent in the simulation tree, the maximum number of searches needs to be performed. In this case, time complexity is $O((log_2(X)+1) \times A)$. Just like SuSeSim, in the CRCB algorithm, the tag cannot be found in the head list of all cache configurations. On the other hand, Janapsatya's method does not support pruning and therefore, for all requests, time complexity of hit/miss determination is fixed to $O((log_2(X)+1) \times A)$. For SuSeSim, CRCB and Janapsatya's method, time complexity for updating linked list is $O((log_2(X)+1)$.

Therefore, considering all the results and complexities, we can say that SuSeSim shows the fastest performance compared to any other method proposed so far for L1 cache.

Bibliography

[1] Xtensa processor. http://www.tensilica.com/.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[3] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[4] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/ markhill/DineroIV/, 2004.

[5] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria. A design framework to efficiently explore energy-delay tradeoffs. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 260–265, New York, NY, USA, 2001. ACM.

[6] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBN System Journal*, 9(2):78–117, 1970.

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21:703–746, 1999.

[8] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[9] K. Horiuchi, S. Kohara, N. Togawa, M. Yanagisawa, and T. Ohtsuki. A data cache optimization system for application processor cores and its experimental evaluation. In *IEICE Technical Report, VLD2006-122, ICD2006-213*, pages 19–24, 2006.

[10] A. Janapsatya, A. Ignjatović, and S. Parameswaran. Finding optimal l1 cache configuration for embedded systems. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 796–801, Piscataway, NJ, USA, 2006. IEEE Press.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *In International Symposium on Microarchitecture*, pages 330–335, 1997.

[12] S. Leibson and J. Massingham. Flix: Fast relief for performance-hungry embedded applications. Technical report, Tensilica Inc., 2005.

[13] X. Li, H. S. Negi, T. Mitra, and A. Roychoudhury. Design space exploration of caches using compressed traces. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 116–125, New York, NY, USA, 2004. ACM.

[14] J. J. Pieper, 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.

[15] D. Ponomarev, G. Kucuk, and K. Ghose. Accupower: An accurate power estimation tool for superscalar microprocessors. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 124, Washington, DC, USA, 2002. IEEE Computer Society.

[16] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comput. Syst.*, 13(1):32–56, 1995.

[17] N. Tojo, N. Togawa, M. Yanagisawa, and T. Ohtsuki. Exact and fast l1 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.

[18] 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.