Highly Efficient LRU Implementations for High Associativity Cache Memory

T.S.B. Sudarshan, Rahil Abbas Mir, S.Vijayalakshmi
Birla Institute of Technology and Science, Pilani, Rajasthan 330331 INDIA
tsbs@bits-pilani.ac.in , rahilabbasmir@rediffmail.com , viji@bits-pilani.ac.in

***Abstract*-High associativity with replacement policy as LRU is an optimal solution for cache design when miss rate has to be reduced. But when associativity increases, implementing LRU policy becomes complex. As many advance and demanding technologies like multimedia, multithreading, database and low power devices running on high performance processors in servers and work stations use higher associativity to enrich performance, there is a need for designing highly efficient LRU hardware implementations. This paper gives analyses various implementations of the LRU policy for a cache with high associativity. The implementation problems are explored, objectives of the design are identified and various implementations namely Square Matrix, Skewed Matrix, Counter, Link-list, Phase and Systolic Array methods are compared with each other on the basis of objective outlined. These implementations are synthesized to determine the constraints and the effect of increase in associativity on the performance. When the associativity is smaller, reduction of associated logic is important and at higher associativity conservation of space is more important. At higher associativity Linked List, Systolic Array and Skewed Matrix are the designs found suitable for implementations.**

I. Introduction

Modern processors, commercial systems, high performance servers, workstation have high associative caches for performance improvement [15,16,17]. The complexity of implementation of LRU (Least Recently Used) policy for highly associative cache tends to increase as the associativity increases [1,2,3,4,10]. The increase in complexity additionally increases the delay incurred to detect the line for replacement. The cache performance is degraded even though a highly associative cache with LRU policy is used due to inapt implementation. This paper analyzes
and compares various efficient LRU implementations for higher associative caches. These designs are analyzed with respect to their implementation complexity and how fast can they determine the replacement cache line. The various implementation of LRU are simulated and synthesized for comparison. The rest of the paper is organized in the following manner. Section 2 identifies higher associativity with LRU as best configuration to reduce miss ratio. Section 3 discusses the implementation complexity of LRU as associativity increases. Section 4 examines various implementations, their working and their characteristics. Section 5 explains the methodology followed to test the functional correctness of the design, and evaluation of the performance metric and the results obtained. Section 6 details the comparison of various implementations based on the results obtained the conclusions are explained in Section 7.

II. Higher Associativity With LRU Policy

The classical approach to improve the cache behavior is reducing miss rate. Increasing associativity in the cache reduces conflict misses thereby reducing miss rates and improving performance. Studies have shown that conflict miss reduces from 28% to 4% when the associativity changes from 1-way to 8-way [2]. Another result showed number of cache misses reduced from 30,000 to as low as 5000 when a higher associativity (512 way) cache is used instead of direct mapped [10]. Further higher associative cache is more efficient when miss penalty is large and memory inter connect contention delay is significant and sensitive to the cache miss rate [6]. Increasing Associativity with any replacement policy often decreases the miss ratio. Better performance of higher associativity depends on efficient replacement algorithm [4]. The replacement algorithm LRU, that replaces the least used line in cache, has miss ratio and performance comparable to optimal (OPT or MIN) algorithm. In LRU policy the line not referenced for the longest period of time is considered as dead line and removed from cache.
LRU is currently the most common replacement strategy used in cache, which gives higher performance [8]. Result from [12] have shown for many workloads FIFO and random yield similar performance but the miss ratio of LRU is 12% lower on the average thus yielding better performance than other policies. Studies [11] have shown that in the case of larger associativity LRU can be noticeably improved and made more optimal when compared to the off-line MIN [7] or the equivalent OPT algorithms [13]. A high associative cache with LRU is a better solution for reducing miss rate and improving performance. This combination has an added advantage of reducing thrashing provided that associativity value, N is greater than M, where M is the different blocks that map to the same set [6]. Results from [23] reveal that cache design affects the behavior of database application and higher associativity gives better performance for database workload. Increasing associativity in Network processor cache removes the problem of cache conflicts [24] enhancing performance. Y. Markoskiy and Y. Patel [20] identifies one of the technique to c-slow a processor is to increase associativity, because higher associativity is useful in providing huge threads, to limit thrashing in multithreading. Higher associativity is reasonable way to increase the physically addressed cache size for it does not increase the translation hardware [21]. Higher associativity

also improves execution time of numerical intensive applications.

III. Implementation Complexity

A 2-way set associative cache with LRU policy can be Implemented with one bit called the access bit. When a line is accessed from the set the access bit of the accessed line is set representing most recently used line and the access bit of the other line is reset to zero representing least recently used line. If associativity is increased to four LRU then it could be implemented as a counter where the number of access bits will be two. Beyond this implementing a LRU policy becomes complex. The number of lines in a cache set increases increasing the storage space to maintain the LRU history, thereby increasing cache size and cost [1,2,3,10]. Complexity of the logic to implement the LRU also increases [4]. Studies reveal that the performance of the LRU policy reduces as the associativity increases [14]. Results obtained shows that the LRU performs close to OPT replacement algorithm when associativity is less but the LRU has a large number of victims to choose from when the associativity is large decreasing performance [13].

Although LRU is shown to be the best replacement policy, which can help to reduce miss ratio and miss penalty yet it performs poorly due to inefficient implementation [4,6]. Efficient implementation of LRU in a high associative cache will increase the performance of the cache. It is shown in [11,4] that for FIFO and random policies the complexity of implementation is relatively low whatever the associativity. Sukumar and others [11] gave a simpler implementation for FIFO and Random. Similarly Round robin is easier to implement since it is only updated on a miss rather than at every hit [2]. Though LRU is the best policy, designers of embedded microprocessors for low power design chose Round robin instead of LRU and made a compromise on performance in order to have simpler implementation [22,9].

An efficient LRU implementation to improve the performance is necessary but implementation has many design constraints. LRU hardware should maintain a data structure where it logs every access to the cache. As the associativity increases the size of the data structure and associated logic also increases. But the storage size cannot be large due to space and time constraints. When the storage space is reduced, the complexity of logic needed to log the access usually increases. Further time taken to log every access and time to find the line to replace when a miss occurs should be less in order to reduce the miss penalty. LRU hardware, with less storage space to log the access, with less complexity in circuit, less time to log the access and less time to detect the replacement line on miss is required for improved performance of cache.

	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

Fig. 1. 4x4 matrix initialized to zero

	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Fig. 2. 4x4 matrix with cache line 3 as the least recently used line

IV. Design Of Various Implementations

The information of each access should be logged in a data structure that determines the performance of the LRU hardware. Each set in the associative cache has its own LRU hardware for implementing the LRU policy. On referencing this set the corresponding hardware is also invoked requiring no separate detection. The collection of this hardware for all the sets in the cache is the Global Set. And the hardware for the set, which is being referenced, is the Working Set. The cache line index in case of hit is index of line whose tag matches with tag bits of referenced address and in the case of miss is the index of line which is identified as the line to replace by LRU hardware. Here we compare six different implementations of LRU policy for attaining high performance for an N-way set associative cache with Square Matrix, Skewed Matrix, Counter, Link list, Phase and Systolic Array methods

A. Square Matrix Implementation

This scheme implements a simple data structure with a simple storage element, D flip-flop. Data structure is a square matrix of this storage element and is of order N for an N-way set associative cache. The global set contains M replications of this data structure for a cache with M sets. Here each of the N rows of the data structure maps to one of the N cache lines of the set and logs the access information of that line. Initially all the bits in matrix are set to zero (Fig. 1). When the cache set is identified, the corresponding data structure in LRU hardware becomes the working set. The Square-Matrix implementation follows a simple logging scheme wherein, it sets the row of accessed line to one and after this sets the column of the accessed line to zero. The number of ones in each row is an indication of the order of the accesses cache lines within the set. A line with more number of 1's is more recently accessed than the one that has less number of 1's. The row in the matrix, which has maximum number of 1's, is the line most recently used and row, which has the entire row set to zero is the line least recently used. On a cache miss, LRU is detected by checking the row for which all the storage elements are zero (Fig. 2). There will always be a line that has the entire row set

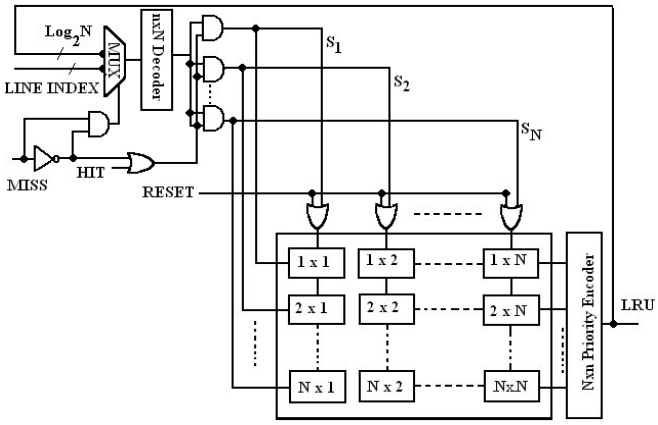

Fig. 3. Square Matrix Implementation

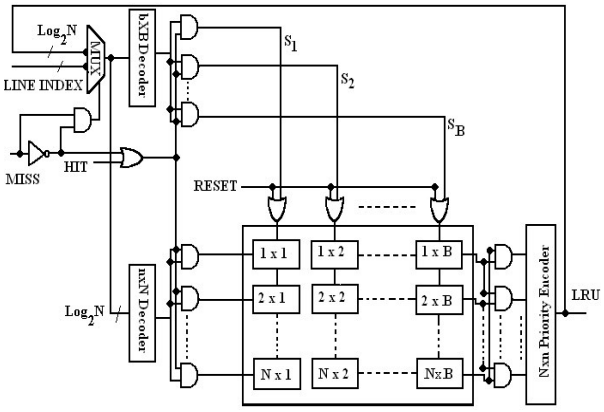

Fig. 5. Skewed Matrix Implementation

to zero. Fig. 4 shows the storage element as an edge-triggered D flip-flop with asynchronous clear, preset which is set when its row is accessed, provided the column of the flip-flop is not accessed at the same time and reset when its column is accessed. The Matrix is made up of N x N such storage elements. The hardware also has one n x N decoder, a 2 to 1 n-bit multiplexer and N x n priority encoder, where n is log2 N. The encoder gives priority to lines with lower index value. Fig. 3 shows the LRU implementation of N-way set-associative cache. The cache line index is presented to the n x N decoder from the multiplexer, which is switched by the hit signal to accept it. The cache line index selects the correct corresponding row and column. The storage elements in the row are set and in the column are reset. The ANDed output of the values of the elements of each row is fed into a priority encoder, which detects the rows whose all elements are Zero and selects one amongst them as the LRU.In case of a miss this index is presented to the multiplexer, which is triggered by the miss signal to accept it and the corresponding row and the column are set and reset respectively. RESET pulse high initializes the matrix by setting all storage elements to zero. The hit or miss decides how the matrix information is to be altered. As very simple operations of set/reset are done on the basic storage elements so that delay involved and time required to log the access is less when there is a hit. The replacement line is obtained from the priority encoder after the values in all the storage elements are ANDed causing considerable delay in the detection of the replacement. The matrix also needs to be updated with the replacement as the index.

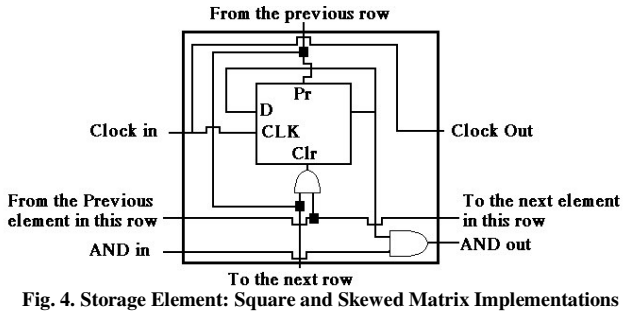

Fig. 4. Storage Element: Square and Skewed Matrix Implementations

Data structure implemented is simple and a minimum of associated logic is required. But the design does not scale well because large amount of space required holding the information that increases quadratically with N, the associativity.

B. Skewed-Matrix Implementation

Skewed-Matrix method is a variant of the previous implementation where a compromise is made in the amount of LRU history being stored. For large sets only a group of cache lines in the set may be active simultaneously. By not keeping the history of other lines, the performance would be slightly affected. The history is kept for a smaller number of lines B, where B is less than N and needs careful choosing with respect to N, since it is a compromise between the accuracy of prediction and storage space for the history. A large value of B will have problems similar to the Square Matrix implementation and a small value would lead to increase in the miss rates.

If a line is not accessed in the last B accesses of the set it is considered to be the least recently used line. So when B is less than N we have more than one line for replacement simultaneously. Skewed Matrix method differs from the previous implementation in choosing the column to be set to zero and in the choice of the replacement row. Rows are set as in Square matrix method but since the number of columns is less so more than one line maps to one column.. N lines clear B columns and so after B accesses more than one row would have row set to zero as N mod B lines would map to the same column. The storage element in the Matrix is the same as explained for Square Matrix method but the Matrix itself has B columns and N rows (Fig. 5). A separate bxB decoder with lower order b lines from the Multiplexer as input, where b is log2B, is used for the columns. The replacement mechanism chooses the row that is zero only if the one above it is not, so as to use all the rows, which would not be possible with the previous implementation. Such rows are checked using the AND gates and then fed into the priority encoder as before giving the replacement line. Skewed-Matrix needs less storage

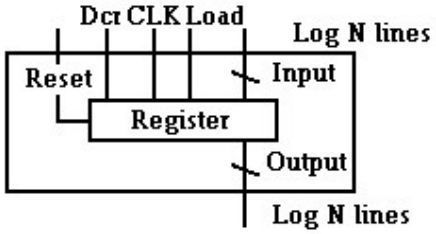

Fig. 6. Storage Element: $\log_2 N$ bit Register

space than the Square Matrix though there is some increase in the complexity of the associated circuitry and the path to detect the replacement. It performs as well as the Square- Matrix implementation given the correct value of B but to predict the correct value of B is difficult.

C. Counter Implementation

Using a register for each row to maintain the LRU history can significantly reduce the large space required by the Square Matrix and Skewed Matrix implementation. As the value of N becomes higher there is exponential drop in the storage space required as compared with previous implementations. There is one to one mapping between the registers, used to record LRU information and the cache lines in a set. The values in the register indicate the order in which the cache lines within a set have been accessed. A register with a larger value means that corresponding cache line is more recently accessed than the line whose register has a lesser value. The smallest value, Zero in the register indicates the corresponding cache line is least recently accessed line and the highest value, N-1 indicates the corresponding cache line is most recently accessed line. Initially all the registers are set to zero. The value of the register, which is called active register whose cache line being accessed is compared with the value of other registers .The registers whose value is greater than active register are decremented and the active register is set to highest value N-1. Counter implementation uses an edge triggered log2N-bit register as a storage element is shown in Fig. 6. The register can be reset to zero, decremented and loaded externally. Each cache line in every set is mapped to a register. The hardware implementation for this data structure for a set (Fig. 7) needs one 1xN log2N-bit demultiplexer, one 2x1 log2N-bit multiplexer, one 1xN 1-bit demultiplexer, one Nx1 log2N-bit multiplexers, N log2N-bit comparators, N log2N-bit registers and one Nx1 priority encoder. The comparator hardware determines the registers whose value is greater than the register of the indexed cache line and equal to it. A zero register means that the line can be replaced and there can be a number of registers that can be used as replacements so a priority encoder decides which of the lines can be reduced. The register is decremented if comparators for that register signals that the register is greater than indexed line provided the load signal for that register is not high. The indexed register is set to N-1 by using the input from first demultiplexer and the correct register to be used for comparison is indicated by the second demultiplexer and is fed to all comparators using the multiplexer. The 2x1 multiplexer is used to select the indexed

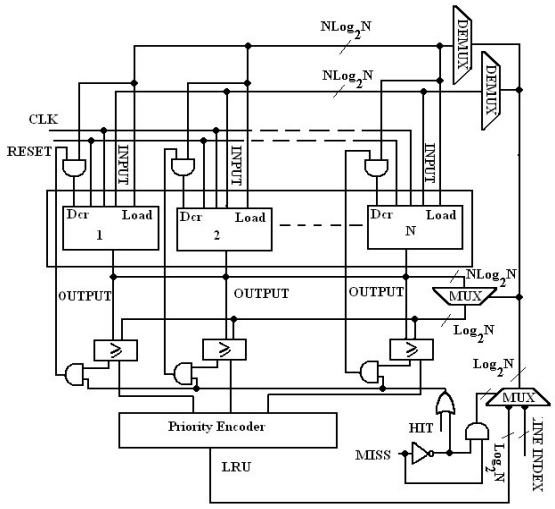

Fig. 7. Counter Implementation

line, which is the replacement line in case of a miss or the accessed cache line index in case of a hit. This implementation uses the minimum, N number of Storage elements, among the various implementations but the associated logic to detect LRU and logging information is more as compared to other implementations. The implementation does not scale well, as the complexity of the associated circuitry increases with N.

D. Phase Implementation

Phase implementation uses the Matrix to implement the Phases concept. The phase is a period where series of reference is made in the set. This implementation is an adaptation from [4] by Yannick Deville and Jean Gobert, which shows that using phases improves the miss ratio. The rows are indexed from 0 to E-1 and the columns from 0 to B-1. E is associativity of the cache and B is free parameter chosen depending on the design but it should be less than and multiple of E. The Matrix is set to zero initially. The column with the highest index is the active phase, so there is no need of the B-pointer, which is a pointer that points to the active phase to track this phase. A counter, E-counter, is used to keep track of the number of lines that have entered the phase and when a maximum of E/B lines are in the phase, new phase starts. The change of phase is indicated by a shift in the Matrix. All the rows of the Matrix are shifted left by one element. At the start of the phase all the highest Index elements are set to zero. Every time a line is accessed its row is set to 1. E-counter is incremented only when a line, which is accessed, has a zero in the highest indexed column of the corresponding row. When the E-counter reaches E/B value, the phase ends. The LRU will be the row that has the least number of ones or the maximum number of zero's. There can be more than one such row. Phase implementation uses a shift register as a basic storage element (Fig. 8).

A shift value of 1 shifts the values in the registers to left, 2 shifts the values in the registers to right and 0 does not shift at

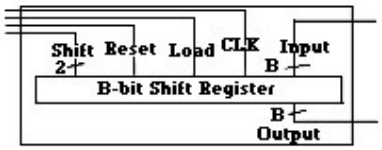

Fig. 8. Storage Element Phase implementation

all. When there is a left shift the MSB is set to zero. The register is set to input when load signal is high and is cleared at reset signal. Along with the N storage elements, the hardware consist (Fig. 9) of a 1 x N Multiplexer, 2 X 1 log2 N bit demultiplexer, a priority encoder, N comparators and a E/B-bit counter, as the E-counter. Each storage element corresponds to the row of the Matrix described previously in this section. When there is an access and the value E-counter is not E/B-1 then the E-counter is incremented provided the MSB (active bit) of the accessed row is not one. An MSB value of the row accessed indicates that it is previously accessed and the counter should not be incremented. If the MSB of the accessed row is 0 and the E-counter has reached its maximum value then the rows are left shifted and the counter is set to 1. On every access, all the bits of the corresponding register of the accessed line are set to 1. Multiplexer selects the accessed row, which is the index in case of a hit or the LRU row in case of a miss. A priority encoder indicates the LRU row.

Phase implementation is similar to Square Matrix or Skewed Matrix implementations but requires more complex logic. Design parameter B has to be chosen carefully. A small value means less storage space but a low accuracy of prediction, whereas a large value means it requires large storage space. Appropriate value of B will help reduce the complexity of the circuit. Accessing a line in Phase implementation takes less time.

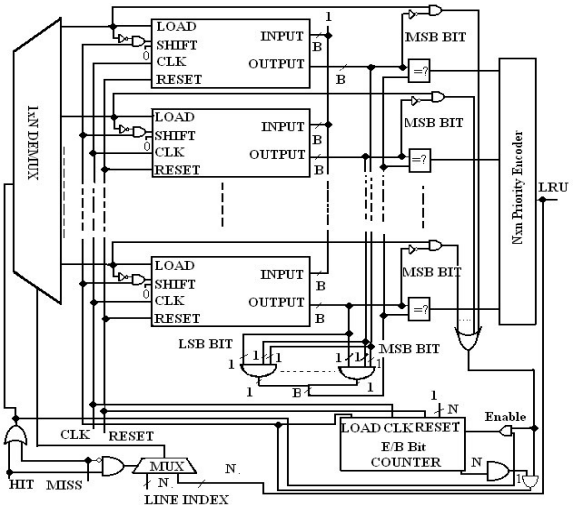

Fig. 9. Phase Implementation

Previous	Next
X	2
1	3
2	4
3	x

1 MRU
4 LRU

Fig. 10. Entry in the Previous list and Next list

E. Link List Implementation

The Link-list implementation, we always know the index of the line to be replaced which incurs considerable delay. The implementation uses less space and uses the logic to determines the LRU line with minimum delay and at the same time update the data structure. The cache line indexes are mapped to two lists Previous and Next. The Next register of the cache line maintain the index of the line that was accessed after that cache line and the Previous register of the cache line maintain the index of the line that was accessed before that cache line. The most recently used cache lines are moved to the head of the list and the less recently used lines to the end of the list. LRU register keeps track of the index at the end of the list and the MRU, the index at the head of the list. An arbitrary ordered list can be chosen to initialize all the registers (Fig. 10) and the line at the end of the list becomes the LRU. When lines are accessed their order of access is determined in the list. The algorithm used for the updating the lists handles the three cases. If the accessed line is LRU then it is made to point to the next of LRU, and if it is the MRU then nothing needs to be done. But for any other line the previous node is made to point to the next node in the Next list and similarly the Previous list is updated. The accessed line is made the MRU.Each of the storage elements in Fig. 11 is basically a log2N-bit register that stores the value when load signal is high. The register is also loaded at reset. X is the hardware that determines the Next or the Previous index value of the line index to which this register in the list is mapped. The hardware looking into the three cases for a set (Fig. 13) has four 1xN log2N-bit demultiplexers, four 1xN 1-bit demultiplexers, two Nx1 log2N-bit multiplexers, one 2x1 log2N-bit multiplexer, N storage elements for the Next list and Previous list each, and two storage elements for LRU and MRU form the LRU hardware. The demultiplexers select the storage element in the other list and also give the value to be stored in this list. The multiplexer selects the correct storage element to which the data must go and also selects the load of that element. Two pairs of multiplexers are used for updating the list with values from the LRU and MRU storage elements and also from the storage elements in the list itself simultaneously. The three cases are handled by the load signal to the two lists from the comparators, which compare the LRU and the MRU with the accessed line index.

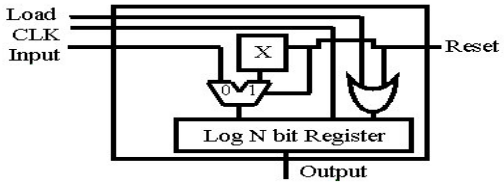

Fig. 11. Storage Element Next List

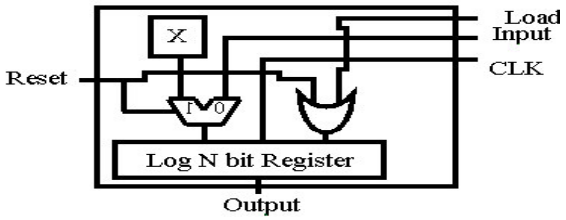

Fig. 12 .Storage Element Previous List

The number of components in the associated logic for this implementation does not increase as the value of N increases, however the size the components increase. But the delay in determining the LRU is not affected much by the increase in the value of N.

F. Systolic Array Implementation

The list for all previous implementations determines the true order of access of the respective cache lines immediately after access but the systolic array, which is an adaptation from the [9] by J.P. Grossman. does not update the list immediately. This scheme has a Systolic node (Fig. 14) which is self comprised consisting of both storage and processing capability. Systolic Array implementation for one set shown in Fig. 15 uses N/2 such nodes to form the list with the last node having the input connected to the output. The LRU is always correct since it is the first node of the list and updated first. Systolic array, which is a version of the link list concept, sorts the line index from LRU to MRU. Each node (Fig. 14) comprises of 3 log2N bit register one for storing the cache line index L that is being accessed, one for current index that is stored in the node and one for storing the index that would be stored in the register at the end of second clock pulse. For sorting, the cache line index L that is accessed is given to the working set. L value passes through each node of the array and is compared with the current indices till the match is found when the Match bit of the node is set to one. The forward

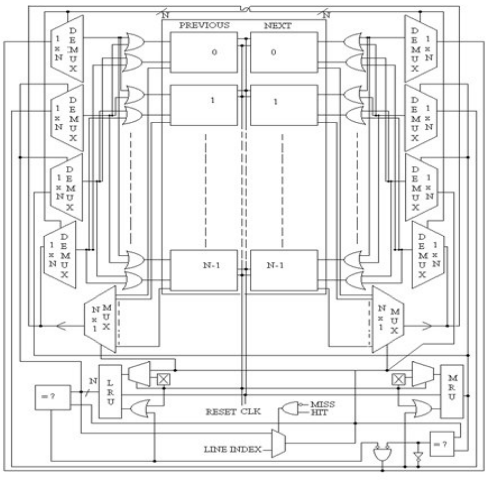

Fig. 13. Link List Implementation

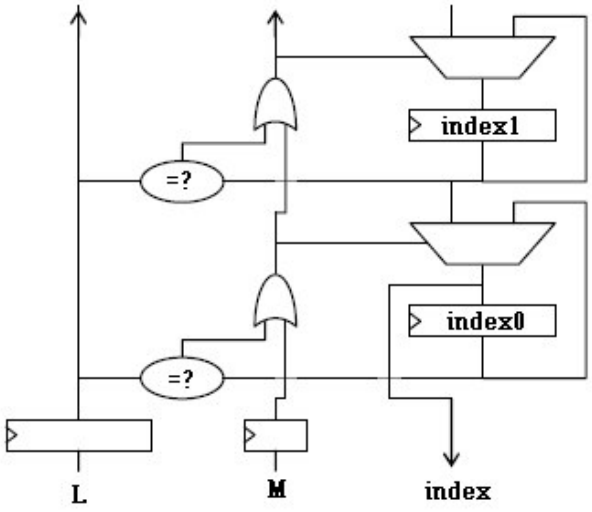

Fig. 14. Systolic Node

signal between two adjacent nodes carries L index and M bit and backward signal carries the current index of the node to the backside node when match M bit is set. The last node is wired to itself for L value to get deposit in the Last node, as it is MRU. The list slowly updates over many cycles but maintains the information correctly. When there is a cache miss, the line for replacement is value in the current index of the first systolic node of the working set and the line index is fed back for updating the access. This reduces the time required for searching the line for replacement. The hardware implementation is accommodated for one cache line access per cycle. This is obtained by the use of two index registers, one log2N bit register for storing current index, one single bit register to store match M, a log N bit multiplexer, a log N bit comparator to compare the L value and current index and OR gate. And every node is made to share the L register, which is again log2N bit.

V. EXPERIMENTAL RESULTS

A. Methodology

The Simulations carried out established the functional correctness of the various LRU hardware implementations. Each implementation was simulated with the cache for associativity of 2,4,8,16,32.Cache size of 2KB, with line size of 32 bytes, word size 32 bits, using write-back and write allocate policies was the design configuration. Active-HDL 6.2 from Aldec, Inc. was used for simulation. The syntheses of different implementations were carried out using the FPGA Advantage 5.2, Leonardo Spectrum Level 3 v2001_1d.45, from Exemplar Logic Inc. The library used for the synthesis is ASIC SCL05u library with ±5 Volt and 300C design parameters. Both Simulation and synthesis was done on a IBM compatible PC with Intel Pentium 4 processor, 256MB RAM, 80 GB Hard disk Space, and Windows XP Professional Edition as the Operating System.

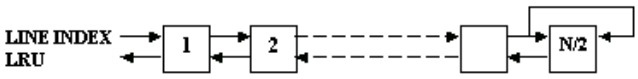

Fig. 15. Systolic Array Implementation

B Result

The graph in Fig. A.1 shows the variation of the number of gates per cache set with the associativity. Fig. A.2 shows the Storage size that the different implementations occupy in the entire cache. The cache considered is a 128KB cache with line size of 32 words. Each word is 4 bytes The design parameter, B for Skewed and Phase implementations used is equal to the associativity when the associativity is less than 16 and equal to 16 for larger associativities. Square Matrix, Skewed Matrix, Counter and Systolic Array show better results and consume less space when the associativity is smaller but at higher associativity Link List, Systolic Array and Skewed Matrices perform better. The graph in Fig. A.3(a) and Fig. A.3 (b) shows the growth of the area with increase in Associativity. 2-way Set Associtivity is taken as the reference and the ratio of the number of gates of all associativites with that of 2-way Set Associative is plotted. Fig. A.3 (a) shows the results per cache set and Fig. A.3 (b) shows the result for entire cache. It can be observed that the growth rates are not uniform for various implementations although the growth rates increase for all implementations. The number of gates for one cache line with change of associativity is plotted in the graph of Fig. A.4. It follows the same trend as the Fig. A.1. The response curves in the Fig. A.1 and Fig. A.4 for Phase and Skewed implementations is because after 16-way associativity the design parameter B, differs from associativity N. Based on the trends of the size of hardware for associativities ranging from 2 to 32, Fig. A.5 gives the projections for a 128KB cache.

VI. Performance Evaluation

The gradients in graph of Fig. A.1 and Fig. A.2 are not smooth indicating that the different implementations do not provide us similar results and behave differently as associativity changes. When the associativity is small all the implementations have more or less the same storage to log information but the small difference in area occupied arises due to difference in the complexity of logic. The associated logic is the dominating factor determining the area occupied for small associativity. From the graphs we can infer that the Square Matrix method occupies the largest area followed by Skewed Matrix, Counter, Systolic, Phase and Link list implementations. The comparison of Skewed Matrix and Square Matrix clearly indicates that at higher associativity storage space must be the criterion that decides the area required. From Fig A.5 it is observed that for 128-way set associativity Square Matrix uses 3 times as many gates as Systolic Array, 2.2 times as many gates as Link List, 1.6 times as many gates as Counter implementation. Hence, for high associativity the implementations that score well are Link List, Skewed Matrix and Systolic methods, which conserve the storage space. Systolic array has the least area requirement as it has small storage space and also does not employ too much logic to update the list quickly. The counter implementation that has the least storage space for the data requires a large area suggesting the fact that reducing the space alone for higher associativity would not provide good results. The associated logic requirement should not to be neglected. This can be observed in Fig. A.4 where the number of gates for the counter increases more rapidly than for Systolic Array or Link List implementations. The gradient in Fig. A.1 is larger than in Fig. A.2 for all the implementations implying that the growth of size for individual sets as associativity increases is more when compared to the growth of size for entire cache. The growth for Matrix and Phase implementations indicate that conservation of space is important to avoid excessive growth but the Counter implementation points out that the growth for the implementations that have least storage space is also not favorable. Skewed and Phase implementations have same characteristics as they use the same storage area for information although the associated logic is different. Link List and phase implementations have the least growth in area. For link list the size of the components involved increases rather than the number of components and for the phase implementation the number of components increases but the size remains the same. The LRU implementations that involve smaller storage space with little increase in component size or number of components show better behavior with increasing associativity. The size of the hardware gives some indications to the delay involved. As the associativity increases the size of different implementations increase indicating that associated delay to retrieve the LRU cache line also increases. The amount of increase in delay for Link List, Systolic and Phase implementations is smaller as the increase in number of gates with increase in associativity, is much slower as compared to other implementations.

VII. Conclusions

This paper has focused on the implementation of the LRU replacement policy for caches with high associativity. As implementing LRU policy in hardware for high associativites is difficult, implementation objectives are identified and various implementations namely Square Matrix, Skewed Matrix, Counter, Phase, Link List and Systolic Array were studied. The results of the different implementations for increasing associativity were analyzed. It is inferred that for higher associativity conservation of space to store data of the schemes is important but the associated logic cannot be totally neglected. At higher associativity link List, Systolic Array and Skewed Matrix are the designs most suitable for implementations, and also with increase in associativity the Link List, Systolic and Skewed Matrix methods would involve less delay. Although the implementation size for one set grows rapidly for increase in associativity, the similar increase when the entire cache is considered is much less. The results also show that the LRU implementations, which involve smaller storage space with little increase in component size or number

of components, show better behavior with increasing associativity. Finally of all the implementations, Systolic and Link List showed better results.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, California: Morgan Kaufman, 2003.

[2] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design*, 2nd ed. San Francisco, California: Morgan Kaufman, 1998.

[3] K. Hwang, A*dvanced Computer Architecture: Parallelism, Scalability and Programmability.* New York: McGraw-Hill Book Co., 1993.

[4] Yannick Deville and J. Gobert, "A class of replacement policies for medium and high-associativity structures," *ACM SIGARCH Computer Architecture News*, vol. 20, no.1,pp. 55-64, March 1992.

[5] Michael zhang and Krste Asanavio, "Highly-Associative Caches for Low-Power Processors," Kool Chips Workshop, in *33rd International Symposium On Micro architecture*, Monterey, California, December 2000.

[6] Chenxi Zhang, Xiaodong Zhang, and Yong Yan, "Two Fast and High-Associativity Cache Schemes," *IEEE Micro Magazine*, vol. 17, no. 5, pp. 40-49, September-October 1997.

[7] Belady, L. A., "A study of Replacement Algorithms for a Virtual-storage Computer," *IBM Systems Journal,* vol. 5, no. 2, pp. 78-101, 1966.

[8] A.J Smith, "Cache Memories," *ACM Computing surveys*, vol. 14, no. 3, pp. 473-500, September 1982.

[9]J..P. Grossman, "A Systolic Array for Implementing LRU Replacement," Project Aries Technical Memo ARIES-TM -18, AI Lab,M.I.T., Cambridge, MA, March 13, 2002.

[10] Brian Harrington, "Cache Memory Overview," http://harr*ingtonweb.com:8090/~brh/misc/*

[11] Sukumar R.A and Abraham S.G, "Efficient simulation of caches under Optimal replacement with application to miss characterization," in *Proceedings of the ACM SIGMETRICS conference on Measurement and modeling* of computer system*, pp. 24-35, May1993.*

[12] J. E. Smith and J. R. Goodman," Instruction cache replacement policies and organizations," *IEEE Transactions on Computers*, vol. C-34, no. 3, pp. 234–241, March 1985.

[13] R.L. Mattson, J. Gecsei, D.R. Slcitz and I.L. Traiger, "Evaluation techniques for storage hierarchies," *IBM System journal pages*, pp. 169-193,1996.

[14] Wayne A. Wong and Jean-Loup Baer, "Modified LRU policies for improving second-level cache behavior," in *Proceedings of the 6th International Symposium on High- Performance Computer Architecture (HPCA)*, pages 49– 60, Toulouse, France, January 2000.

[15]"Can a server platform cost -effectively power your enterprise?,"*http://www.intel.com/*business/bss/swapps/server2003/xeon_server 2003.pdf.

[16]"PowerPC 440GP Embedded Processors," "*http://www306.ibm.com/chips/techlib/techlib.nsf/techdocs/9C268609D3538C 4687256A33004E124B/$file/440GP_pb.pdf.*

[17] "Universal Platform System-on-Chip Processor, Entertainment Processors: EP9312,"*http://www.cirrus.com/en/products/pro/detail/P131.html*.

[18] Mark Papermaster, Robert Dinkjian, Michael Mayfield, Peter Lenk, Bill Ciarfella, Frank O'Connell and Raymond DuPont , "POWER3: Next Generation 64 -bit PowerPC Processor Design," *http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/power3wp. pdf*, October 7, 1998.

[19]"Excalibur Device Overview," *http://www.altera.com/literature/ds/ds_arm.pdf*

[20] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek," Post Placement C-slow Retiming for the Xilinx Virtex FPGA ," in Proce *edings of the Eleventh ACM International Symposium on Field-Programmable Gate Arrays* (FPGA 2003), Monterey CA, Feb. 23--25, 2003.

[21] Prof. Philip Koopman, *18-548/15-548 - Memory System Architecture, Associativity*, Carnegie Mellon University, fall semester, 1998.

[22] Lawrence T. Clark, Eric J.Hoffman, Jay Miller, Manish Biyani, Yuyun Liao, Stephen strazdus, Michael Morrow, Kimberley E.Velarde and Mark A.Yarc, "An Embedded 32b microprocessor core for low power and high performance applications," *IEEE journal of solid state circuits*, vol. 36, no. 11, pp 1599-1608, November2001

[23] Anastassia Ailamaki," ARCHITECTURE- CONSCIOUS DATABASE SYSTEMS," Ph.D. dissertation,Univ. of Wisconsin — Madison, Computer Sciences Dept., Wisconsin, 2000.

[24] Kartik Gopalan and Tzi-cker Chiueh, " Improving Route Lookup Performance Using Network Processor Cache", in *Proc. of SC2002 High Performance Networking and Computing*, Baltimore, MD, November 2002.

[25] Robert E Moncrief II ,"IBM RS/6000," http://www.cs.jmu.edu/users/abzugcx/CS480&585/Term-Projects/IBM- RS-6000-by-Robert-Moncrief2002SUMMER-CS-585.doc,July 15,2002

[26] "Leonardo Spectrum User's Manual," Software Version v2001.1, August 2001

[27] Dr. Scott F. Smith, *EE 433/533-Embedded and Portable Computing Systems, ARM CPU Cores*, College of Engineering, Boise State University, spring semester, 2004

[28]"SA-110 Microprocessor, Technical Reference Manual" "ftp://download.intel.com/design/strong/manuals/2780 5801.pdf,December 2000.

Appendix:

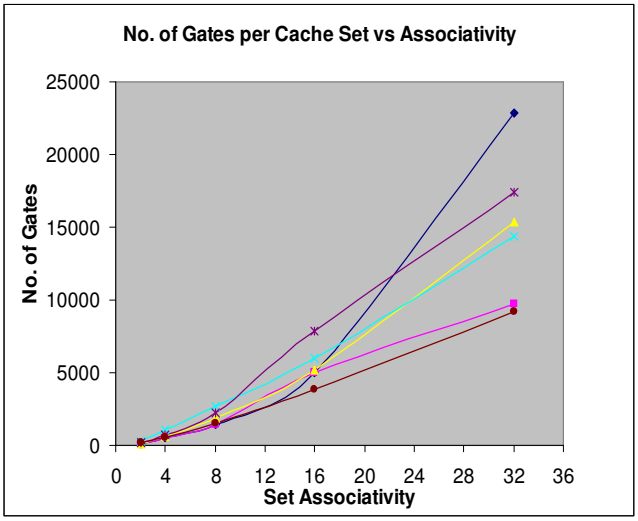

Fig. A.1. No of Gates per cache set vs Associativity

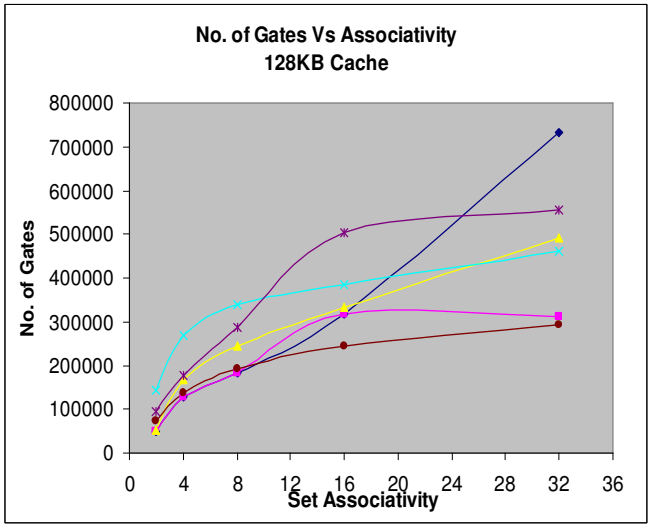

Fig. A.2. No of Gates vs Associativity for 128KB cache

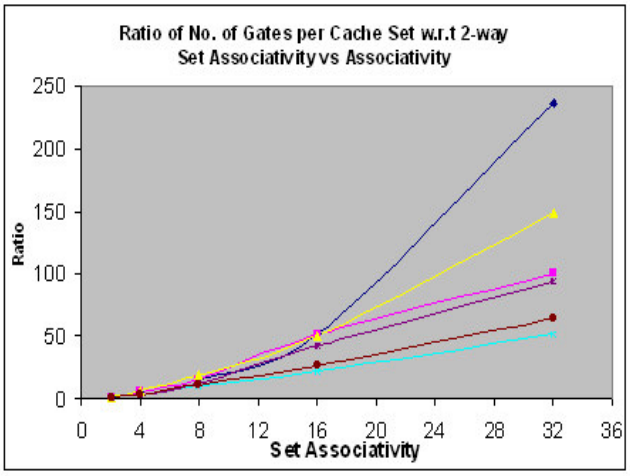

Fig. A.3(a). Ratio of No. of Gates per Cache Set w.r.t 2-way Set Associativity vs Associativity

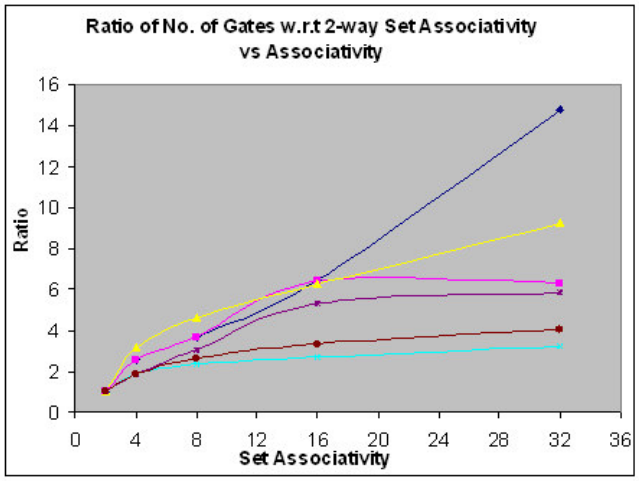

Fig. A.3(b). Ratio of No. of Gates w.r.t 2-way Set Associativity vs Associativity

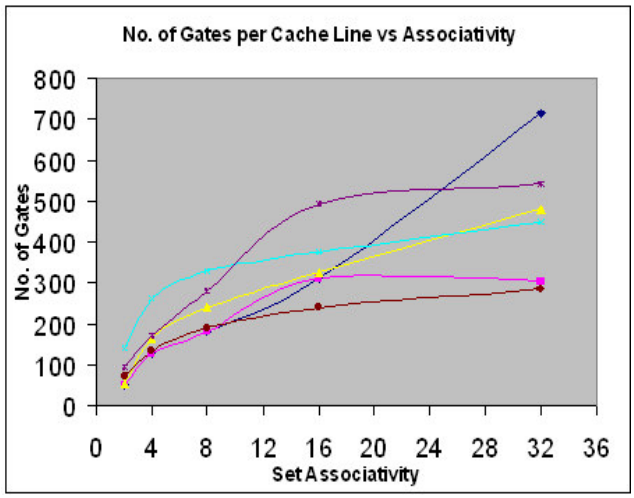

Fig. A.4. No. of Gates per Cache Line vs Associativity

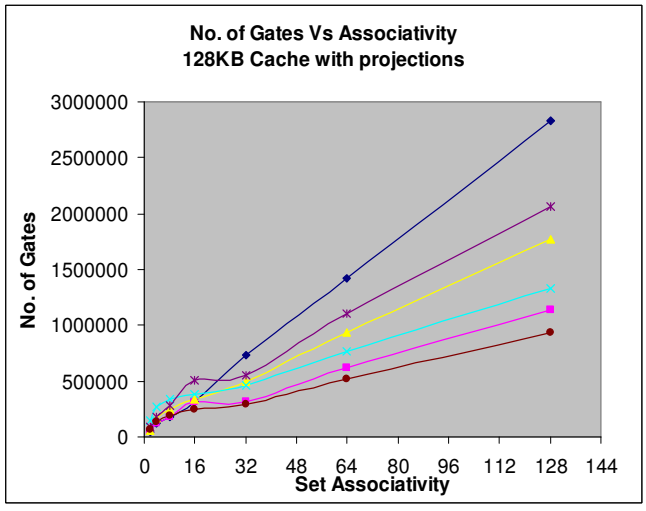

Fig. **A.5. No. of Gates vs Associativity for a 128 KB cache (projections)**