Least Recently Plus Five Least Frequently Replacement Policy (LR+5LF)

Adwan AbdelFattah[1] and Aiman Abu Samra[2]
[1]Computer Science Department, The Arab American University of Jenin, Palestine
[2]Computer Engineering Department, The Islamic University of Gaza, Palestine

Abstract: *In this paper, we present a new block replacement policy in which we proposed a new efficient algorithm for combining two important policies Least Recently Used (LRU) and Least Frequently Used (LFU). The implementation of the proposed policy is simple. It requires limited calculations to determine the victim block. We proposed our models to implement LRU and LFU policies. The new policy gives each block in cache two weighing values corresponding to LRU and LFU policies. Then a simple algorithm is used to get the overall value for each block. A comprehensive comparison is made between our Policy and LRU, First In First Out (FIFO), V-WAY, and Combined LRU and LFU (CRF) policies. Experimental results show that the LR+5LF replacement policy significantly reduces the number of cache misses. We modified simple scalar simulator version 3 under Linux Ubuntu 9.04 and we used speccpu2000 benchmark to simulate this policy. The results of simulations showed, that giving higher weighing to LFU policy gives this policy best performance characteristics over other policies. Substantial improvement on miss rate was achieved on instruction level 1 cache and at level 2 cache memory.*

Keywords: *Cache memory, replacement policy, LRU, LFU, miss rate.*

Received January 29, 2010; accepted September 20, 2010

1. Introduction

Caching is the main method used to decrease the speed gap between processor and main memory [9, 13]. There are three basic cache organizations. First, direct mapping, where any block has a unique place in cache and no need to replacement policy. This implementation has a good hit time but worse miss rate. Second, the fully associative, which allows a memory block to be mapped to any of the empty cache blocks, but if there is no empty blocks a replacement policy used to evict one from all over the cache. This organization is very expensive in hardware implementation and has worse hit time, as all addresses will be compared to get a wanted block.

The third organization is set-associative, which divides the cache into sets and allows a memory block to be mapped at any empty block within a set. If there are no empty blocks, a replacement will evict a block from this set only. This organization is a trade-off between the previous two organizations. As it is trade-off between costs, hit time and miss rate.

One of its most important design decisions is the block replacement policy. Effective replacement policy is the topic of much research in computer systems. All cache algorithms have one common function; which is to reduce the miss rate [1, 3, 13]. The cost of misses includes miss penalty, power consumption, and bandwidth consumption. The "hit rate" of a cache describes how often a searched-for block is actually found in the cache. The choice of a block replacement algorithm, in set associative caches, can have a great effect on the overall system performance [3, 4]. More efficient replacement policies keep track of more used blocks in order to improve the hit rate for a given cache size. Each replacement strategy is a compromise between hit rate and latency.

Some policies are trivial others are complex. Trivial policies are the first approaches used to determine the replacement candidates. They include the Random replacement where the block to be replaced is selected randomly from all the blocks in the set. Another is the FIFO replacement, where the set is designed as a queue structure and every block that is inserted to the set will be ordered according to the time they were inserted in the cache. These trivial policies results in higher miss rate. A miss is the failure to find a required block in the cache and hence it must be requested from the main memory. There is a number of popular policies that depend on two measures. One is the Recency which is the time span from the current access time to the last access time of a certain block. Recency was exploited by Least Recently Used (LRU) policy and its different implementations. The second measure is the Frequency, which was exploited by its basic implementation, Least Frequently Used (LFU) policy. In this policy, each cache block maintains a counter that is incremented each time the block is accessed. When it is required to pick a candidate for replacement, the block with the minimum counter value, that means minimum frequency, is picked. These two extremes were used excessively in the cache replacement policies research [7, 17].

LRU ignores the usability of the block so the most accessed block may be the victim. LFU ignores the latest accessed block, so the latest block may be the victim, and may not take the chance to increase its value. So the challenge was to combine frequency and recency to obtain expectedly better performance in terms of hit ratio.

In this paper, we present a new block replacement policy, which combines LRU and LFU. Experimental results show that our proposed replacement policy, Least Recently Plus five Least Frequently (LR+5LF), significantly reduces the number of cache misses.

We used Simple Scalar simulator version3 under Linux Ubuntu 9.04 to get the results of the comparisons between our proposed algorithm and other popular algorithms. We modified the routine code related to the replacement algorithm in the Simple Scalar simulator.

The Simple Scalar tools set is a system software infrastructure used to build modelling applications for program performance analysis, detailed micro architectural modelling, and hardware-software co-verification. Using the Simple Scalar tools, users can build modelling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. The Simple Scalar tools are used widely for research and instruction, for example, in 2000 more than one third of all papers published in top computer architecture conferences used the Simple Scalar tools to evaluate their designs. In addition to simulators, the Simple Scalar tools set includes performance visualization tools, statistical analysis resources, debug and verification infrastructure [12].

We used spec2000 benchmark to simulate this policy. The results of simulations showed, that giving higher weighing to LFU policy gives this policy best performance characteristics over other policies. Substantial improvement on miss rate was achieved on instruction level 1 cache and at level 2 cache memory.

2. Related Work

The study of block replacement policies is, in essence, a study of relating past access patterns with future access behaviour. Based on the Recognition of access patterns through acquisition and analysis of past behaviour or history, replacement policies resolve to identify the block that will be used furthest down in the future, so that that block maybe replaced when needed [2, 8].

The LRFU policy associates a value with each block. This value is called the Combined Recency and Frequency (CRF) value and quantifies the likelihood that the block will be referenced in the near future. Each reference to a block in the past contributes to this value and a reference's contribution is determined by a weighing function F(x), where x is the time span from the reference in the past to the current time [7](Figure 1).

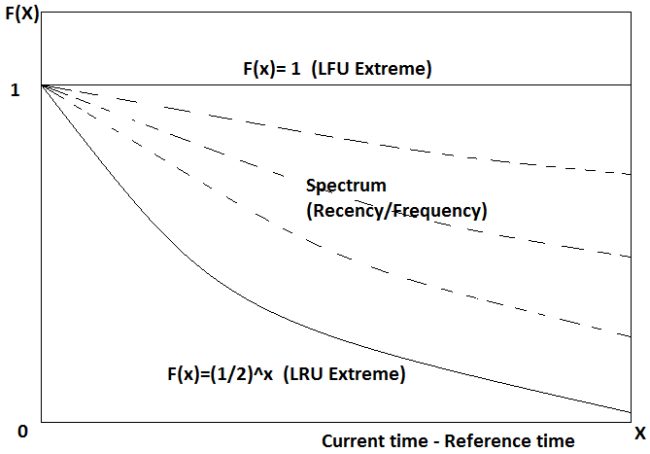

Figure 1. LRFU spectrums.

Zhansheng *et al*. [17], proposed a novel replacement policy that switches between LRU and LFU on runtime, they used a queue called Qout as a separate data structure within the cache. The queue is limited in size and the replacement is done using LRU in this queue. Whenever a block is removed from the cache, it is placed in Qout. They used an additional H (hit) counter that is initialized to zero and an O (out) counter that is also initialized to zero. The cache is initially managed by LRU policy and with each miss, the old block will be pushed to Q and the new block will be checked to find if it already exists in Q. if it does, then the H counter is incremented by one and if not the O counter is incremented by one. The replacement decision is made by comparing the values of H and O, if the value of H is larger than Q then the policy will be switched to LFU and if O is larger than H, the policy will be changes to LRU and so on. This method is an adaptive method and the idea is that when the program is first started and the cache still has free lines, then the recency will be more likely to affect the replacement decision. And once the value of H exceeds O then this indicates that the blocks that were used before are needed and hence the frequency becomes more dominant than the recency and vice versa when O exceeds H. The implementation of this scheme is complex which is negatively affect performance.

Other polices combines LRU and LFU in one scheme. They used different implementations but they are trying to optimize the performance of cache replacement algorithms [15, 16]. Some others use Dynamic Insertion Policy (DIP), in which the selection of replacement policy depends on which one incurs fewer misses [10].

3. LR+5FU Policy

LR+5FU: is a novel replacement policy which is a combination between two popular replacement policies LRU and LFU. During the development of the proposed policy, the following problems have been solved:

- LRU and LFU weighing.
- Combining of LRU and LFU.
- Determining the line to be replaced.

3.1. LRU and LFU Weighing

To obtain a balance between two policies we add a weighing value to each of them by developing the following weighing algorithms.

3.1.1. LRU Weighing Algorithm

The weighing value changed from <0> (least recently/frequently used) to <associative-1> (most recently/frequently used).

Algorithm 1 LRU Weighing

1. *Hit block in cache*
2. *For(i=0;i<assoc;i++)*
3. *If block.wlru <Wlru[i]*
4. *Wlru=wlru-1*
5. *End for loop*
6. *Block.wlru=assoc-1*

Where wlru[i] is a weighing value of LRU. When the block in cache is accessed, the LRU algorithm gives this block the Most Recently Used value, so the algorithm works as follow: check weighing values for all blocks in the set; if any block has a weighing value larger than the weighing value of accessed block, reduce it by 1, finally put the weighing value of accessed block as the largest number which equals to associative-1.

The implementation of this algorithm is simple and needs a little hardware to be added, and it is done during transferring of the word to CPU.

In Figure 2, we show a transition state diagram which describes hardware implementation of the suggested algorithm for 4-way associative cache. Any hit to block will change the state of the block to highest weighing value and decrease others, h1 means block in location one has been hit; the states from S0 to S3, is the LRU weighing values of lines (0, 1, 2, 3) for four ways set associative mapping.

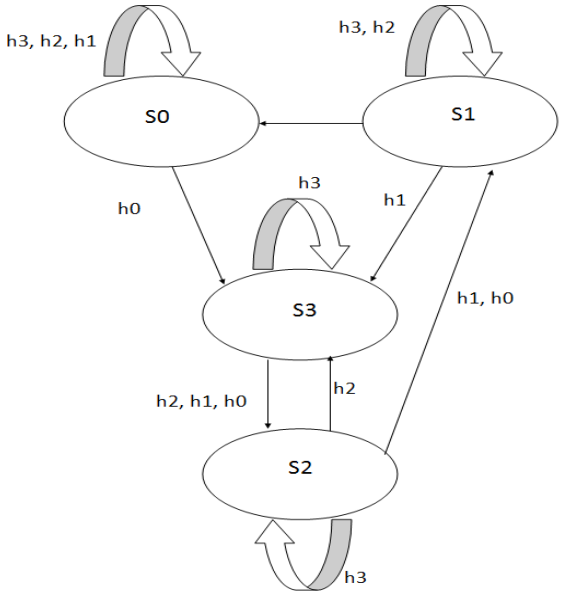

Figure 2. LRU transition states diagram.

3.1.2. LFU Weighing Algorithm

This algorithm produces conversion of the values in counters corresponding to each line, to the range of weights that has been used in LRU algorithm.

Algorithm 2 LFU Weighing

1. *LFU [n] is an array that contains the values of counters for all lines.*
2. *LFUW[n] is an array of weights for all lines.*
3. *n- number of lines in the set .*
4. *LFUW [n] = {0}*
5. *For (i = 0; i<n-1; i++)*
6. *For (j = i + 1 ; j< n; j++)*
7. *IF (LFU [i]> LFU [j])*
   LFUW [i]++;
8. *Else If (LFU [i]< LFU [j])*
   LFUW [j]++;

3.2. Combining Weighing Value

Replacement policy depends on the Weighing Least Recently Least Frequently Used (WLRFU) values, which determined by the following equation:

$$WLRFU[i]=WLRU[i]*Cr + WLFU[i]*Cf \quad (1)$$

Where Cr and Cf: are priority constants for LRU and LFU respectively. In simulating process we will use the values of these constants to improve the performance.

3.3. Determining the Line to be Replaced

In normal case the line with minimum WLRFU will be replaced, but in case, where two or more blocks have the same WLRFU value as shown in Table1, for four ways set mapping, algorithm 3 should be used to resolve this problem.

The following algorithm has been developed to be flexible to different systems requirements and this

achieved by using priority constants, which can be assigned according to the priority of latency or frequency in a specific systems.

Table1. Possible WLRFU for four ways set mapping.

LRU_c or LFU_c	$LRU_c + LFU_c$
L_{c0}	0 , 1 , 2 , 3
L_{c1}	1 , 2 , 3 , 4
L_{c2}	2 , 3 , 4 , 5
L_{c3}	3 , 4 , 5 , 6

Algorithm 3 Determining the Line to be Replaced

1. *sum= LRU_c + LFU_c*
2. *IF (sum = 3) then Replace L_{c2}, or L_{c1} → depend on the priority constants.*
3. *Else IF (sum = 2) then Replace L_{c1}*
4. *Else IF (sum = 1) then Replace L_{c1} or L_{c0} → depend on the priority constants.*
5. *Else IF (sum = 0) then Replace L_{c0}*

Where LFU_c or LRU_c is a least frequently or least recently used counter respectively, L_{ci} - the line which it's LFU_c or LRU_c= i, from Table 1 we see that L_{ci} line should be selected carefully to be the median of LFU_c and LRU_c, and by using this constraint the least miss rate has been achieved.

4. Additional Hardware

Hardware implementation is simple, we need two counters for each block in the cache; the first is for LRU weighing and the other is for LFU weighing. If the memory organization is four-way associative then these counters are two bits length, if the cache organization is eight-way associative then these counters are three bits length.

In addition to the mentioned above hardware, we need a large counter to store the usage number of each block plus small storage for Cr and Cf. Note that if the original hardware architecture uses the LFU algorithm then the additional hardware is easy to add and implement, it is just full adder plus shift register.

5. Simulation and Results

In this section, we discuss the results from trace-driven simulations performed to assess the effectiveness of the proposed LR+5LF policy. We modified the simple scalar v3 simulator to implement the LR+5LF replacement policy. We used benchmark SPECCPU2000. The SPECCPU2000 benchmark suite is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers. The benchmarks in this suite were chosen to represent real-world applications, and thus exhibit a wide range of runtime behaviours [14].

We compared the LR+5LF policy with other policies that depends on cache size, multilevel caches, block size and associativity by using the following benchmarks:

1. *Gcc*: Integer component of SPECCPU2000, C language optimizing compiler, 176.gcc is based on gcc version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled [14].
2. *Vpr*: Integer component of SPECCPU2000, Integrated Circuit Computer-Aided Design Program (More specifically, performs placement and routing in Field-Programmable Gate Arrays) [14].
3. *Parser*: Integer component of SPECCPU2000, Word Processing, The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words [14].
4. *Equake*: Floating point component of SPECCPU2000, The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin. The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method [14].
5. *Vortex*: Integer component of SPECCPU2000, VORTEx is a single-user object-oriented database transaction benchmark which exercises a system kernel coded in integer C. The VORTEx benchmark is a derivative of a full OODBMS that has been customized to conform to SPECCINT2000 (component measurement) guidelines [14]. Figure 3 shows the relationship between cache size, memory organization and miss rate, at 1M level 2 cache on 2-ways, 4-ways, 8-ways and fully associative mapping, it is clear that the performance is approximately identical for all types.

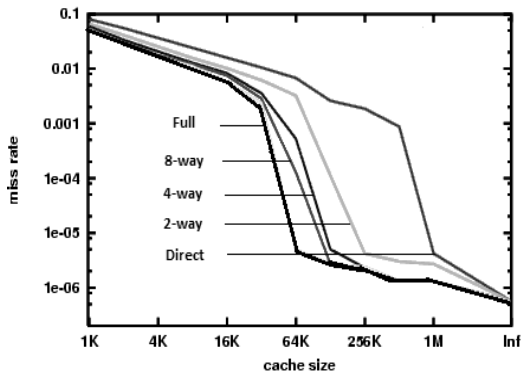

Figure 3. Miss rate versus cache size on the Integer portion of SPECCPU2000.

Assigning Values to *Cr* and *Cf* Constants: As shown in Figure 4, the equation 1 gives the highest performance results at *Cr=1* and *Cf=5*. As we found that giving *Cf* value larger than 5 doesn't increase the performance. This is why our policy called (LR+5LF). The formulation of this equation is similar to reducing λ value in [1], to improve performance, where reducing λ means the behaviour of LRFU policy is becoming closer to LFU rather than to LRU.

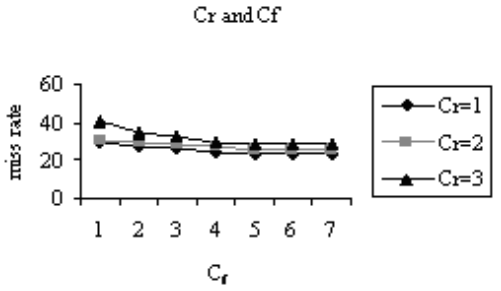

Figure 4. The effects of Cr and Cf values on miss rate.

Our simulation focused on *L2* cache, so we used the same configuration of *L1* cache for all policies, in which a separate instruction *L1* and Data *L1* cache with 128 sets, 32 bytes block size and 4-way associatively has been used for each of them.

Figures 5 and 6 show the miss rate of LR+5LF policy compared to LRU and FIFO by using four different benchmarks. It is clear that the suggested LR+5LF policy gives better results in the miss ratio than both LRU and FIFO in L2 cache. In simulation running time we found that the LR+5LF policy achieves good improvement in miss rate at instruction level 1 cache, but no change in data level 1 cache over other algorithms.

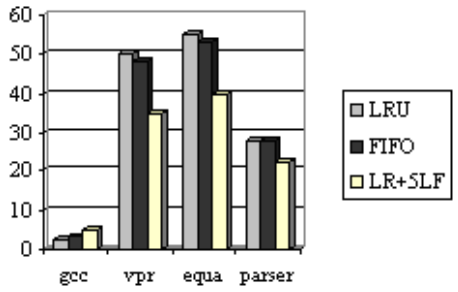

Figure 5. Miss rate at L2 with cache 1024 sets and 64-bits block size and 4-way associativity.

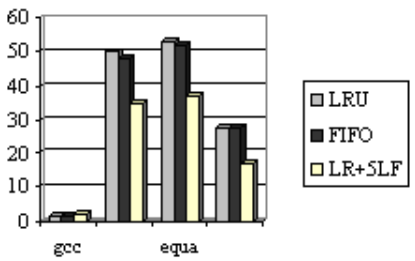

Figure 6. Miss rate at L2 cache with 2048 sets and 64-bits block size and 4-way associativity.

In Figure 7, we compared our policy with three important policies. First one is the optimum theoretical policy in which LRU is self-correcting according to the past accesses by adding a shadow directory and a mistake history table. When there is a miss the OPT [5, 6] will choose either to replace one of the blocks in the cache or choose not to replace the missed block and bypass it. Second one is V-way policy which try to reduce conflict misses done in each set associative in L2 cache, in which the tag are doubled and the replacement becomes globally across L2 cache [11]. Third one is CRFP which is a self-tuning and can switch between different cache replacement policies adaptively and dynamically in response to the access pattern changes [17].

The comparison was done by using vortex benchmark (object oriented database) from SPECCPU2000. We found that the LR+5LF policy gave the best results in miss rate. It was the closest to optimum theoretical.

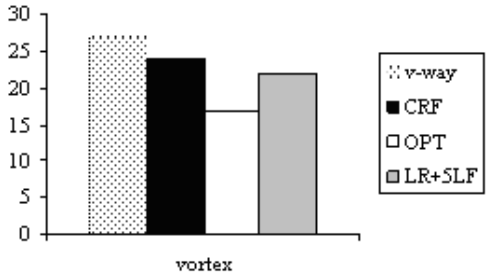

Figure 7. Comparison of LR+5LF with other policies using 1M level 2 cache.

6. Conclusions

We have developed a new policy (LR+5LF) which is based on combining LRU and LFU replacement policies in an efficient manner. To assess the effectiveness of the proposed LR+5LF policy, we have used the simple scalar v3 simulator to implement the proposed replacement policy in addition to Gcc, Vpr, Vortex, Parser and Equa benchmarks from SPECCPU2000 benchmark suite. We show the relationships between cache size, memory organization and miss rate, at level 2 cache according to 2-ways, 4-ways, 8-ways and fully associative mapping which seems approximately identical for all types. Constants Cr and Cf can be easily scaled in order to achieve the highest performance results. For our proposed policy, they should be assigned 1, 5 values respectively. By using different benchmarks the suggested LR+5LF policy gave better results in the miss ratio than both LRU and FIFO in L2 cache.

We found that the LR+5LF policy achieved good improvement in miss rate at instruction level 1 cache and Identical to other algorithms at cache data level 1. The LR+5LF policy has been compared with other

policies like optimum theoretical policy, v-way cache, and CRF using vortex benchmark (object oriented database) from SPECCPU2000.We found that the proposed policy gave the best results in miss rate. It was the closest to optimum theoretical.

References

[1] Alghazo J., Akaaboune A., and Botros N., "SF-LRU Cache Replacement Algorithm," *in Proceedings of International Workshop on Memory Technology Design and Testing*, USA, pp. 19-24, 2004.

[2] Belady A., "A Study of Replacement Algorithms for Virtual-Storage Computers," *Computer Journal of IBM Systems*, vol. 5, no. 2, pp. 78-101, 1966.

[3] Hennessy J. and Patterson D., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, 2007.

[4] Jamil T. and Stacpoole R., "Cache Memories," *IEEE Potentials*, vol. 19, no. 2, pp. 24-29, 2000.

[5] Kampe M., Stenstrom P., and Dubois M., "Self-Correcting LRU Replacement Policies," *in Proceedings of the 1st Conference on Computing Frontiers*, USA, pp. 181-191, 2004.

[6] Kaushik R. and Govindarajan R., "Emulating Optimal Replacement with Shepherd Cache," *in Proceedings of the 40th International Symposium on Microarchitecture*, Chicago, pp. 445-454, 2007.

[7] Lee D., Choi J., Kim J., Noh S., Min S., Cho Y., and Kim C., "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transaction on Computers*, vol. 50, no. 12, pp. 1352-1361, 2001.

[8] Mattson L., Gecsei J., Slutz R., and Traiger L., "Evaluation Techniques for Storage Hierarchies," *Computer Journal of IBM Systems*, vol. 9, no. 2, pp. 78-117, 1970.

[9] Megiddo N. and Modha D., "ARC: A Self-Tuning, Low Overhead Replacement Cache," *in Proceedings of the 2nd USENIX Symposium on File and Storage Technologies*, USA, pp. 115-130, 2003.

[10] Qureshi K., Aamer J., Yale N., Patt C., Steely J., and Joel E., "Adaptive Insertion Policies for High-Performance Caching," *in Proceedings of the 34th International Symp-osium on Computer Architecture*, USA, pp. 381-391, 2007.

[11] Qureshi K., Thompson D., and Patt N., "The V-Way Cache: Demand Based Associativity via Global Replacement," *in Proceedings of the 32th International Symposium on Computer Architecture*, USA, pp. 544-555, 2005.

[12] Simple Scalar, LLC, available at: http://www.simplescalar.com/, last visited 2010.

[13] Stallings W., *Computer Organization and Architecture*, Prentice Hall, 2006.

[14] Standard Performance Evaluation Corporation**,** available at: http://www.spec.org/, last visited 2010.

[15] Wong A. and Baer L., "Modified LRU Policies for Improving Second-Level Cache Behavior," *in Proceedings of 6th International Symposium on High-Performance Computer Architecture*, France, pp. 49-60, 2000.

[16] Yoon J., Min L., and Cho Y., "Buffer Cache Management: Predicting the Future from the Past," *in Proceedings of International Symposium on Parallel Architecture Algorithms and Networks*, Philippines, pp. 92-97, 2002.

[17] Zhansheng L., Dawei L., and Huijuan B., "CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies," *in Proceedings of IEEE 8th International Conference on Computer and Information Technology Workshops*, Sydney, pp. 72-79, 2008.

Adwan AbdelFattah is an assistant professor at the Computer Science Department of the Arab American University of Jenin, Palestine. Previously, he worked at Philadelphia and Zarqa Private University. He received his PhD from the National Technical University of Ukraine in 1996. His research interests include computer networks, computer architecture, cryptography, networks security, authentication and digital signature.

Aiman Abu Samra is an IEEE and computer society member. He received his PhD from the National Technical University of Ukraine in 1996. Currently, he is an assistant professor at the Computer Engineering Department at the Islamic University of Gaza, Palestine. His research interests include computer architecture, computer networks and software engineering. He managed several funded projects in cooperation with industry. He teaches several courses on computer architecture and computer networks.