Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite

Hussein Al-Zoubi, Aleksandar Milenkovic, Milena Milenkovic
Department of Electrical and Computer Engineering
The University of Alabama in Huntsville
{alzoubh | milenka | milenkm} @ece.uah.edu

ABSTRACT

Replacement policy, one of the key factors determining the effectiveness of a cache, becomes even more important with latest technological trends toward highly associative caches. The state-of-the-art processors employ various policies such as Random, Least Recently Used (LRU), Round-Robin, and PLRU (Pseudo LRU), indicating that there is no common wisdom about the best one. Optimal yet unattainable policy would replace cache memory block whose next reference is the farthest away in the future, among all memory blocks present in the set.

In our quest for replacement policy as close to optimal as possible, we thoroughly explored the design space of existing replacement mechanisms using SimpleScalar toolset and SPEC CPU2000 benchmark suite, across wide range of cache sizes and organizations. In order to better understand the behavior of different policies, we introduced new measures, such as cumulative distribution of cache hits in the LRU stack. We also dynamically monitored the number of cache misses, per each 100000 instructions.

Our results show that the PLRU techniques can approximate and even outperform LRU with much lower complexity, for a wide range of cache organizations. However, a relatively large gap between LRU and optimal replacement policy, of up to 50%, indicates that new research aimed to close the gap is necessary. The cumulative distribution of cache hits in the LRU stack indicates a very good potential for way prediction using LRU information, since the percentage of hits to the bottom of the LRU stack is relatively high.

Keywords
Cache memory, replacement policy, performance evaluation.

1. INTRODUCTION
Cache memories remain one of the hot topics in the computer architecture research, since the ever-increasing speed gap between processor and memory only emphasizes the need for more efficient memory hierarchy. As modern processors include multiple levels of caches, and as cache associativity increases

[8], it is important to revisit the effectiveness of common cache replacement policies. When all the lines in a cache memory set become full and a new block of memory needs to be placed into the cache memory, the cache controller must discard a cache memory line and replace it with the new data from the main memory. Preferably, the discarded cache memory line will not be needed in the near future. However, the cache controller can only guess which cache memory line should be discarded. An optimal replacement (OPT) algorithm would replace a cache memory block whose next reference is the farthest away in the future among all the cache memory blocks presently in the set [2]. This policy requires the perfect knowledge of the future block references, and hence its implementation is infeasible. Instead, heuristics have to be used to determine which block is the most suitable to be replaced.

The state-of-the-art processors employ various policies such as Random [5], LRU (Least Recently Used) [1], Round-robin (or FIFO – First-In-First-Out) [8], and PLRU (Pseudo LRU) [7] indicating that there is no common wisdom about the best cache replacement policy. All these mechanisms, except Random, determine which cache memory block to replace by looking only at the cache memory past references. LRU replacement requires a number of status bits to track when each cache block is accessed. The number of these bits increases as the set-associativity increases. To reduce the cost and complexity of LRU policy, Random policy can be used, but potentially at the expense of performance. Several researchers and computer designers have considered these two heuristics as too extreme in terms of implementation cost and performance. They have proposed various PLRU heuristics to reduce the hardware cost by approximating the LRU mechanism.

Recent studies explore cache design space with relatively limited associativity, and consider only true LRU policy [4], [11], [12]. There have been some efforts to further improve cache replacement decisions, for example using the compiler [13] or hardware/software techniques [14].

The goal of our study is to explore common cache replacement policies in greater depth: we feel that some fundamental questions have not been answered in previous work, or just partially answered. We would like to know what is the performance of different cache replacement policies for contemporary workload, and different cache configurations, how existing policies relate to OPT, does a replacement policy have different effect on instruction and data caches, and how good are pseudo techniques in approximating true LRU.

The performance analysis is based on using SimpleScalar's [3] cache simulators executing SPEC CPU2000 benchmark suite

[6], as a widely used workload for high performance computing. OPT, LRU, Random, FIFO, and two pseudo LRU polices, tree-based and MRU-based, have been studied on a wide range of cache organizations, varying cache size, associativity, and cache level.

The results of our study provide a solid starting point in a research of new cache replacement heuristics for contemporary workload, and support possible use of LRU bits for way prediction, or selective turning off of less used cache ways.

The rest of the paper is organized as follows. Section 2 shortly describes evaluated cache replacement heuristics, and Section 3 describes experimental methodology. Section 4 poses the questions about replacement policies that were the focus of this study, and Section 5 gives the corresponding answers. Section 6 concludes.

2. COMMON CACHE REPLACEMENT POLICIES

The LRU mechanism uses a program's memory access patterns to guess that the cache line which has been accessed most recently will, most likely, be accessed again in the near future, and the cache line that has been "least recently used" should be replaced by the cache controller. An example of how the LRU stack is maintained is shown in Figure 1. Although the LRU replacement heuristic is relatively efficient, it does require a number of bits to track when each block is accessed, and relatively a complex logic. Another problem with the LRU heuristic is that each time the cache hit or miss occurs the block comparison and LRU stack shift operations require time and power.

To reduce the cost and complexity of the LRU heuristic, Random policy can be used, but potentially at the expense of performance. Random replacement policy chooses its victim randomly from all the cache lines in the set. An obvious way to implement it is with a simple Linear Feedback Shift Register (LFSR). Round Robin (or FIFO) replacement heuristic simply replaces the cache lines in a sequential order, replacing the oldest block in the set. Each cache memory set is accompanied with a circular counter which points to the next cache block to be replaced; the counter is updated on every cache miss.

PLRU schemes employ approximations of the LRU mechanism to speed up operations and reduce the complexity of implementation [9], [10]. Due to the approximations, the least recently accessed cache memory is not always the location to be replaced. Here, we will discuss two implementations, a tree-based and a MRU-based. In the tree-based PLRU replacement heuristic *nway-1* bits are used to track the accesses to the cache blocks, where *nway* represents the number of cache blocks (ways) in a set. Figure 2 illustrates tree-based PLRU (PLRUt) using a 4-way cache memory as an example. The track bits B0, B1, B2 form a decision binary tree. The track bit B1 indicates whether two lower cache blocks CL0 and CL1 (B1=1), or 2 higher cache blocks CL2 and CL3 (B1 = 0) have been recently used. The track bit B0 determines further which one of two blocks CL0 (B0=1) or CL1 (B0=0) has been recently used; bit B2 keeps the access track between cache lines CL2 and CL3. On a cache miss, bit B1 determines where to look for the least recently block (2 lower cache lines or 2 higher cache lines). Bit

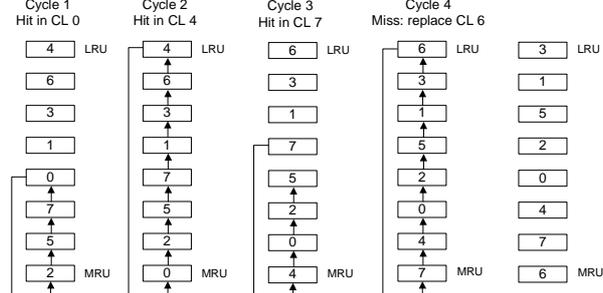B0 or B2 determines the least recently used block. On a cache hit, the tree bits are set according to this policy.

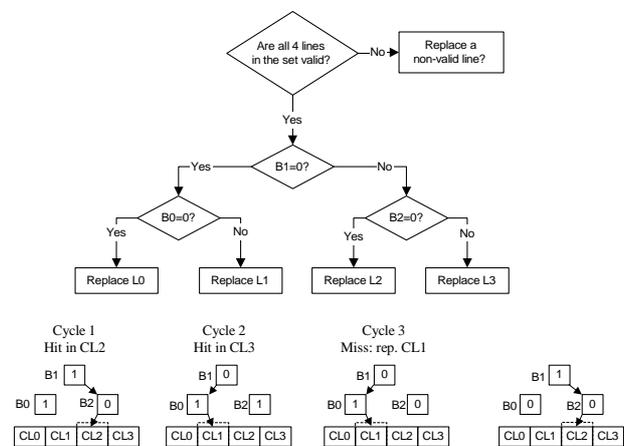
Figure 1. An Illustration of LRU Mechanism.

Figure 2. Tree-based Pseudo Least Recently Used Policy.

The other implementation of the PLRU heuristic is based on using the most recently used (MRU) bits (PLRUm). In this case each cache block is assigned an MRU bit, stored in the tag table. The MRU bit for each cache block is set to a "1" each time a cache hit occurs on the cache block, indicating that the cache block has recently been used. When the cache controller is forced to replace a cache block, it examines the MRU bit for each cache block looking for a "0". When it finds a "0", the cache controller replaces that cache block and then sets the MRU bit to a "1". A problem could occur if the MRU bits for all cache memory blocks are set to a "1". If this happens, all the blocks are unavailable for replacement causing a deadlock. To prevent this type of deadlock, all the MRU bits in the set are cleared except the MRU bit being accessed when a potential overflow situation is detected. An example in Figure 3 illustrates the MRU-based PLRU.
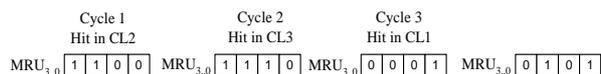

Figure 3. Pseudo Least Recently Used Policy based on MRU bits.

Table 1 gives storage requirements and corresponding actions taken on a cache hit and a cache miss for all replacement

policies discussed. Random policy guarantees the minimal hardware cost, while the LRU hardware cost increases dramatically for caches with associativity larger than 8. In 2-way cache organizations PLRUt policy requires only one track bit, and shows the same performance as other LRU-based polices, hence it is an obvious choice for 2-way caches. For caches with higher associativity, PLRUt and PLRUm have roughly the same complexity. If we take the number of transitions in track bits as a qualitative measure of power consumption and assume the same number of misses, it is clear that PLRUm shows slightly better characteristics than PLRUt, and both are much better than LRU.

Table 1. Complexity comparison of different cache line replacement heuristics

Heuristic	Storage requirements [bits]	Action on cache hit	Action on cache miss
Random	$log_2(ways)$	None	Update LFSR register
FIFO	$nsets \cdot log_2(ways)$	None	Increment FIFO counter
LRU	$nsets \cdot ways \cdot log_2(ways)$	Update the LRU stack	Update the LRU stack
PLRUm	$nsets \cdot ways$	Update the MRU bit(s)	Update the MRU bit(s)
PLRUt	$nsets \cdot (ways-1)$	Update the tree bit(s)	Update the tree bit(s)

3. EXPERIMENTAL METHODOLOGY

Performance evaluation of different cache replacement policies has been done using the sim-cache and sim-cheetah simulators from the Alpha version of the SimpleScalar toolset [3]. The original simulators have been modified to support additional pseudo-LRU replacement policies and collect corresponding statistics. In order to allow tracking of the dynamic behavior of caches, the sim-cache simulator has been modified to print interval statistics per specified number of instructions.

Selected benchmarks from SPEC CPU2000 [6] suite have been used as a simulation workload, representing the state-of-the-art applications for high-performance computing. The initial performance evaluation, based on the sim-cheetah simulator results, has filtered out benchmarks insensitive to increase in cache associativity greater than two ways. For example, 175.vpr is highly sensitive to change in data cache associativity, having a relatively large number of conflict misses, whereas 173.applu does not benefit at all from more than two cache ways.

Selected SPEC CPU2000 integer and floating point benchmarks were used with reference data inputs. First 500 million instructions have been skipped and then 500 million instructions simulated. For each benchmark a number of simulations have been run for various cache organizations - 1-, 2-, 4-, 8-, 16-, and 32-way, replacement policies Random, FIFO, LRU, PLRUt, and PLRUm, and various cache sizes. The first set of experiments concentrates on performance of split first level instruction and

data caches (L1I, L1D) with 4KB, 8KB, 16KB, and 32KB sizes. The second set of experiments considers performance of the second level unified cache memory (L2U) with 32KB, 64KB, and 128KB, assuming direct-mapped 4KB first level caches for both data and instructions. The experiments for first level cache have also been conducted for OPT replacement policy. Although OPT requires a perfect knowledge of future references, and hence cannot be implemented, it is useful as a yardstick in exploring potentials of replacement policies. In all experiments a cache line size is 32 bytes. In order to provide the monitoring of cache miss rate during program execution, cache misses are recorded for each 100000 instructions, as well as for the whole application.

As a measure of cache performance we use the number of misses per 1000 instructions (MP1K). The relationship between this measure and more traditional cache miss rate is shown in (1).

$$MP1K = 1000 \times MissRate \times \frac{MemoryAccesses}{InstructionCount} \qquad (1)$$

Additional experiments measure the cumulative distribution of cache hits in the LRU stack. The purpose of these experiments is to support the future research of cache power management and way prediction [6] based on exploiting replacement policy.

4. CACHE REPLACEMENT QUESTIONS

While a lot of research has been done in the field of cache memories, some important questions about cache replacement policies applied to the state-of-art workload still remain without complete answers. In this section we list the well-known observations about replacement policies, along with related questions to which our study offers answers.

1. "Cache associativity reduces the number of cache misses." Question is how much associativity is enough for the state-of-the-art benchmarks? Is there for some benchmarks a point of diminishing returns? These questions are partly answered [4], [11], but only for LRU replacement policy.

2. "There is a significant gap between OPT policy and LRU, indicating a potential for further improvement of replacement efficacy." Exactly how large is this potential? We want to see exactly how much space there is for improvement for each specific benchmark and cache configuration. Related work does give some answers [14], but the results differ and usually just the average value is given.

3. "LRU usually performs better than Random and FIFO". What is the exact relationship between cache miss rates produced by these policies for a wide range of benchmarks and cache configurations? Do they behave differently for different types of memory references, such as instruction and data?

4. "Cache miss rate with a policy A for benchmark B and cache organization X is Z". Yes, but what can we say about dynamic cache misses, during program execution? If some policy is on average better than others, does it stay consistently better? Can dynamic change of replacement policy reduce the total number of cache misses?

5. "LRU replacement policy exploits cache reference locality." How often do we hit the last recently used way? What is the average length of a LRU stack accesses? What is the hit

distribution? Can we use Most Recently Used information for cache way prediction?

6. "True LRU policy is expensive, so let's use pseudo LRU instead." How good are pseudo LRU techniques at approximating true LRU? We consider two different mechanisms: the widely known tree-based and the not-so-well-known MRU- based.

5. EVALUATING CACHE REPLACEMENT POLICIES: ANSWERS WE HAVE GOTTEN

In this section we give the answers to the questions raised in the previous section. Due to space restrictions, we do not present the comprehensive simulations results, but show enough to illustrate the observations we made. Full set of results is available at http://www.ece.uah.edu/~lacasa/crp02/crp.htm.

1. For data cache, our experiments show that the most important performance gain is for transition from a direct mapped to a two-way set associative cache. On average, for LRU replacement policy we have 3-14 cache misses less per 1K instructions executed (Table 2 & 3). The reduction is larger for smaller caches. An increase in associativity from 2-way to 4-way reduces the number of misses for up to 5.

For integer applications, increased associativity has a significant performance potential, since for OPT replacement policy the number of cache misses is reduced with an increase in associativity. However, realistic replacement policies often reach the point of diminishing returns at 8 ways for small caches, and 2 to 4 ways for larger caches.

For floating point applications, increasing associativity to more than 4 ways with OPT replacement policy is beneficial only for relatively small caches (4KB, 8KB). For larger caches, associativity higher than two does not significantly reduce the number of cache misses. For realistic replacement policies, higher associativity is beneficial for small caches and some applications, up to 16 ways for 172.mgrid and 183.equake, and 32 ways for 200.sixtrack. For larger caches, there is no need for more than 2 ways.

For instruction cache, OPT replacement policy benefits from increased associativity, while realistic policies for most benchmarks do not successfully exploit more than 8 ways, or in some cases even more than 2 ways.

Realistic replacement policies in unified second level L2U cache are on average more sensitive to increased cache associativity than in first level instruction and data caches, even for up to 32 ways, although some benchmarks, such as 200.sixtrack, do not benefit from more than 4 ways.

2. We observed a very interesting rule of thumb related to OPT replacement policy. OPT performance of a data cache of a certain size is roughly equal to LRU performance of a cache twice as big, with the same number of ways (Table 2 & 3). Figure 4 illustrates this result for application 300.twolf with horizontal arrows from OPT to LRU replacement policy. For example, OPT replacement policy in a 2-way, 4KB cache has 36.85 misses per 1K instructions, while LRU policy in a 2-way, 8KB cache has 35.87 misses. Results from Table 2 & 3 indicate that for data references OPT offers about 20-40% less cache

misses over the best considered non-optimal policy for integer, and about 5-40% for floating point applications, depending on cache size and associativity. The gap is larger for smaller caches due to more conflict misses. Somewhat surprisingly, the significant improvement due to OPT is present even in 2-way caches. The gap between LRU and OPT is even larger for instruction references, indicating that further investigation of cache replacement heuristics to close the gap between OPT and currently used policies could be highly beneficial for both data and instruction caches.

3. As it could be expected, LRU policy in data caches has better performance than FIFO and Random, across almost all evaluated benchmarks and cache sizes. Yet there are some exceptions: for 301.appsi, 253.perlbmk, and 183.equake, Random policy is sometimes slightly better than LRU. Compared to LRU policy, Random is on average about 22% worse, while FIFO is about 20% worse.

In L1 data cache there is no clear winner between FIFO and Random replacement policy, and the difference between the two decreases as the cache size increases. For one group of applications, 172.mgrid, 176.gcc, 197.parser, and 300.twolf, caches with FIFO replacement have less misses for all considered cache sizes and organizations. For other group, 191.fma3d, 186.crafty, and 183.equake, Random always outperforms FIFO. For the rest of the considered benchmarks, for smaller cache sizes FIFO dominates, while for larger caches Random policy is better or same as FIFO.

In instruction L1 caches LRU replacement policy has less cache misses than FIFO and Random for most evaluated configurations, although in some cases, such as 253.perlbmk for 16KB and 32KB caches, Random policy significantly outperforms the other two. FIFO and Random policies behave in similar way for most benchmarks, and Random outperforms FIFO in some cases.

The results for the average L2U cache miss rate are following trends observed for L1D, with LRU outperforming FIFO and Random policies, or being only slightly worse than Random.

4. Although the number of cache misses per 1K executed instructions varies during application execution, if one policy is on average better than the other, our experiments show that it stays consistently better, that is, it may be worse only in relatively short periods of time.

5. The cumulative distribution of cache hits in the LRU stack indicates a very good potential for a way prediction using LRU information, since the percentage of hits to the bottom of LRU stack is relatively high. For example, for 16 ways and cache size 16KB, average percentage of hits to the bottom of the LRU stack is about 70% for integer and 65% for floating point benchmarks. Figure 5 shows data for 300.twolf, and data cache with 16 ways. For a fixed number of ways, percentage of hits to the bottom of the LRU stack increases as the size increases, which can be explained by less contention in larger caches. For a fixed cache size, this percentage decreases as the number of ways increases. Since for most of the benchmarks the point of 90% is reached well beyond the LRU stack size, power management that would turn off less used ways could be beneficial.

Table 2 Average L1D misses per 1000 instructions for 8 SPEC2000 integer applications (175.vpr, 176.gcc, 186.crafty, 197.crafty, 252.eon, 253.perlbmk, 255.vortex, 300.twolf)

		SpecINT L1D misses per 1K inst.					
		1W	**2W**	**4W**	**8W**	**16W**	**32W**
4K	fifo	48.45	37.79	34.93	33.84	33.24	33.17
	lru	48.45	34.92	30.79	29.02	28.22	27.98
	random	48.45	38.60	35.90	34.85	34.23	34.18
	plru-t	48.45	34.92	31.06	29.47	28.84	28.50
	plru-m	48.45	34.92	30.81	28.88	28.05	27.81
	opt	48.45	25.34	20.81	18.65	17.38	16.81
8K	fifo	33.48	24.41	21.64	20.77	20.17	19.79
	lru	33.48	22.57	18.84	17.46	16.65	16.03
	random	33.48	24.64	21.51	20.53	19.99	19.64
	plru-t	33.48	22.57	18.87	17.50	16.84	16.38
	plru-m	33.48	22.57	18.38	17.03	16.24	15.70
	opt	33.48	16.26	12.56	11.16	10.38	9.96
16K	fifo	21.16	15.01	12.60	11.41	10.94	10.80
	lru	21.16	13.93	11.06	9.65	9.09	8.86
	random	21.16	15.06	12.68	11.59	11.19	11.02
	plru-t	21.16	13.93	11.19	9.97	9.69	9.56
	plru-m	21.16	13.93	10.83	9.47	8.94	8.77
	opt	21.16	10.31	7.84	6.76	6.25	6.02
32K	fifo	13.26	9.33	7.95	7.42	7.30	7.26
	lru	13.26	8.66	7.11	6.47	6.31	6.25
	random	13.26	9.47	8.14	7.63	7.45	7.40
	plru-t	13.26	8.66	7.31	6.76	6.69	6.64
	plru-m	13.26	8.66	7.03	6.42	6.26	6.20
	opt	13.26	6.89	5.46	4.83	4.55	4.42

6. In first level instruction and data caches, pseudo-LRU heuristics, PLRUm and PLRUt, are very efficient in approximating LRU policy. PLRUm outperforms PLRUt, and even performs slightly better than LRU, providing the best performance at minimal cost. Pseudo LRU techniques are consistently close to LRU during whole program execution.

For second level unified cache L2U, both PLRUm and PLRUt outperform LRU for even more cache organizations than in first level caches.

6. CONCLUSION

As the speed of processors increases much faster than the decrease in memory latency, eliminating cache misses will continue to be extremely important for improving overall processor performance. With caches becoming more set-associative, cache replacement policies will gain even more significance. This research encompasses a detailed performance evaluation of the common replacement policies using SimpleScalar toolset and the latest SPEC CPU2000 benchmark suite.

Our results show that while OPT replacement policy generally benefits from increased associativity, realistic policies do not

successfully exploit the large number of cache ways. The gap between LRU and OPT replacement policies, up to 50%, indicates that new research aimed to close the gap is necessary. Furthermore, for data caches we observed that OPT policy performance is near the performance of the next best policy of a cache twice as big. Consequently, caches with better replacement policies closer to OPT could be smaller and yet have the same performance as today's caches, thus significantly reducing power consumption.

Table 3 Average L1D misses per 1000 instructions for 5 SPEC2000 floating-point applications (172.mgrid, 183.equake, 192.fma3d, 200.sixtrack, 301.appsi)

|		SpecFP L1D misses per 1K inst.					
		1W	**2W**	**4W**	**8W**	**16W**	**32W**
4K	fifo	49.22	38.22	33.66	27.99	24.84	24.80
	lru	49.22	36.68	31.71	24.96	22.34	22.17
	random	49.22	38.85	37.30	30.95	27.46	27.21
	plru-t	49.22	37.42	31.90	25.25	22.37	22.40
	plru-m	49.22	37.42	31.90	24.70	21.90	21.75
	opt	49.22	25.05	20.28	16.54	14.09	13.43
8K	fifo	27.92	21.77	19.03	17.69	16.67	16.66
	lru	27.92	20.05	16.58	15.08	14.41	14.54
	random	27.92	23.37	21.21	20.04	17.51	16.40
	plru-t	27.92	20.84	17.37	16.11	13.94	14.39
	plru-m	27.92	20.84	17.12	15.32	12.88	13.43
	opt	27.92	15.98	12.22	9.82	8.42	8.11
16K	fifo	14.03	11.35	10.40	11.30	10.01	9.41
	lru	14.03	10.79	9.55	8.83	8.52	8.51
	random	14.03	12.91	12.41	12.90	12.33	11.71
	plru-t	14.03	11.27	9.63	9.81	9.34	9.23
	plru-m	14.03	11.27	9.59	9.91	8.91	8.93
	opt	14.03	9.10	7.95	7.55	7.46	7.43
32K	fifo	11.63	7.96	7.94	7.95	7.97	8.16
	lru	11.63	7.57	7.44	7.43	7.40	7.51
	random	11.63	8.64	8.95	9.08	9.40	9.19
	plru-t	11.63	7.87	7.43	7.37	7.33	7.36
	plru-m	11.63	7.87	7.39	7.38	7.38	7.52
	opt	11.63	7.18	6.92	6.84	6.78	6.74
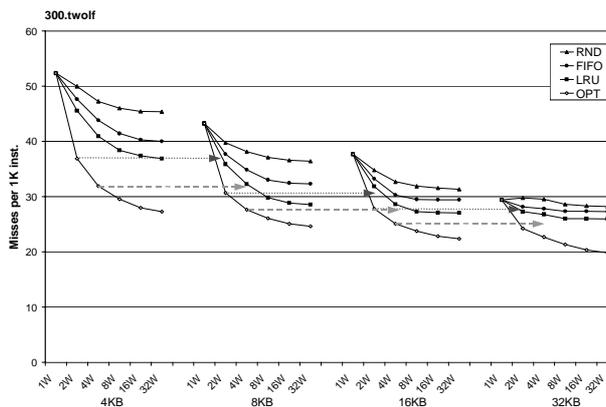
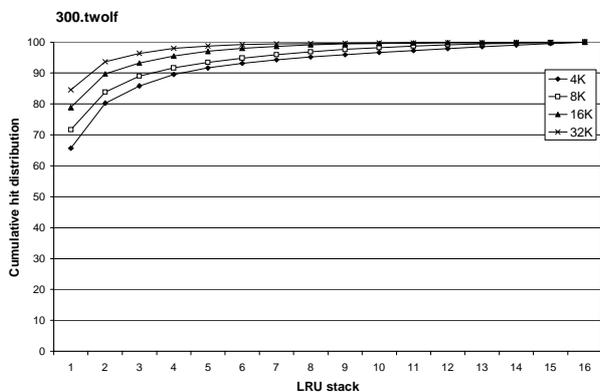**Figure 4. L1D misses per 1000 instructions for 300.twolf**

Figure 5. LRU cumulative hit distribution for 300.twolf

7. REFERENCES

[1] Ackland B., Anesko D., Brinthaupt D., Daubert S.J, Kalavade A.., Knoblock J., Micca E., Moturi M., Nicol C.J., O'Neill J.H., Othmer J., Sackinger E., Singh K. J., Sweet J., Terman C. J., and Williams J., "A Single-Chip, 1.6 Billion, 16-b MAC/s Multiprocessor DSP," *IEEE Journal of Solid-state circuits*, Vol. 35, No. 3, March 2000, pp. 412-423.

[2] Belady, L.A., "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal*, 5(2):79-101, 1966.

[3] Burger D., Austin T., "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Technical Report #1342*, June 1997.

[4] Cantin J. F, Hill M. D., Cache Performance of the SPEC CPU2000 Benchmarks, http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/

[5] Hennessy J. L., Patterson D., *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann Publishers, 2003.

[6] Henning J. L., SPEC CPU2000: Measuring CPU Performance in the New Millennium, *IEEE Computer*, vol. 33, no. 7, July 2000, pp. 28-35, http://www.spec.org/osg/cpu2000/papers/COMPUTER_20 0007-abstract.JLH.html

[7] Intel ® Pentium ® 4 and Intel ® Xeon Processor Optimization – Reference Manual ™ - Reference Manual, http://developer.intel.com.

[8] Intel ® Xscale ™ Core – Developer's Manual, December 2000, http://developer.intel.com

[9] Malamy A., Patel R., Hayes N., "Methods and aparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature," *United States Patent 5353425*, October 1994.

[10] K. So, R. N. Rechtshaffen, "Cache operations by MRU change," *IEEE Transaction on Computers*, vol. 37, no. 6, pp. 700-707, June 1988.

[11] Sair S., and Chamey M., "Memory Behavior of the SPEC2000 Benchmark Suite." *IBM Thomas J. Waston Research Center Technical Report RC-21852*, October 2000.

[12] Thomock N. C, Flangan, J.K., "Using the BACH Trace Collection Mechanism to Characterize the SPEC 2000 Integer Benchmarks," Workshop on Workload Characterization, September 2000.

[13] Wang Z., McKinley K., Rosenberg A., Weems C., "Using the Compiler to Improve Cache Replacement Decisions," *Proceeding of the International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, September, 2002.

[14] Wong W., Baer J-L., "Modified LRU Policies for Improving Second-level Cache Behavior," *Proceedings of the 6th International Symposium on High-Performance Computer Architecture,* Toulouse, France, January 2000