Small-LRU: A Hardware Efficient Hybrid Replacement Policy

Purnendu Das[1], Bishwa Ranjan Roy[2*]
Department of Computer Science
Assam University Silchar
INDIA

***Abstract*—Replacement policy plays a major role in improving the performance of the modern highly associative cache memories. As the demand of data intensive application is increasing it is highly required that the size of the Last Level Cache (LLC) must be increased. Increasing the size of the LLC also increases the associativity of the cache. Modern LLCs are divided into multiple banks where each bank is a set-associative cache. The replacement policy implemented on such highly associative banks consume significant hardware (storage and area) overhead. Also the Least Recently Used (LRU) based replacement policy has an issue of dead blocks. A block in the cache is called dead, if the block is not used in the future before its eviction from the cache. In LRU policy, a dead block can not be remove early until it become LRU-block. So, we have proposed a replacement technique which is capable of removing dead block early with reduced hardware cost between 77% to 91% in comparison to baseline techniques. In this policy random replacement is used for 70% ways and LRU is applied for rest of the ways. The early eviction of dead blocks also improves the performance of the system by 5%.**

Keywords*—Replacement policies; cache memories; last level cache; hardware overheads; dead block

I. Introduction

Replacement policy plays the most significant role in the performance of highly set-associative cache architecture. In multi-level cache, the first level cache (L1) is allotted as private cache to individual core whereas the large last level cache (LLC) is shared by all the cores. To reduce the access latency the LLC is also divided into multiple banks where each bank is a set-associative cache. The data distribution among the banks is based on different data mapping policies [1]. In this work we consider Static Non Uniform Cache Access (SNUCA) where each block has a fixed bank to be mapped and the bank is called the *home-bank* of the block [1]. Fig. 1 shows a multicore processor having 4 cores. Each core has a private L1 cache and a part of shared L2 cache. To make the design simple we have not divided the L2 into more than 4 banks. The rest of the paper follows the same multicore processor as shown in Fig. 1. Each bank is a set-associative cache as shown in Fig. 2.

Today's data intensive applications demand larger and higher associative cache (specially LLC). These highly associative cache reduces the conflict misses and hence improves the performance of the system. But these highly associative cache has some overheads in terms of hardware. One such overhead is because of maintaining replacement policy for such highly associative banks. In set-associative cache (or bank), each set maintains its own replacement policy. This policy is required to replace an existing block from the cache.

As mentioned above, each set in the set-associative cache has its separate replacement hardware. For an N-way set associative cache, each set maintains separate hardware for its replacement policy. To insert a newly incoming block in the set, one of the existing block need to be evicted first known as the *victim block*. The purpose of replacement policy is to select a victim block to replace it with the recently requested block. One of the most popular and well known replacement policy is called Least Recently Used (LRU) policy. The concept of this technique is well known and not necessary to discuss here. To maintain LRU policy in each set having N ways, each way must uniquely represent its age relative to the other ways. For example, if there are only 2 ways in a cache then each set needs only 1 bit to maintain the relative age. Bit-0 means old and bit-1 means new. Similarly 2 bits are required to uniquely maintain the relative age in case of a 4-way set associative cache. Hence for a N-way set associative cache, $log_2 N$ bits are required to maintain the relative age of each ways in the set. The details about the hardware overheads required to maintain replacement policy is discussed in Section II.

The three important operations of any replacement policy are:

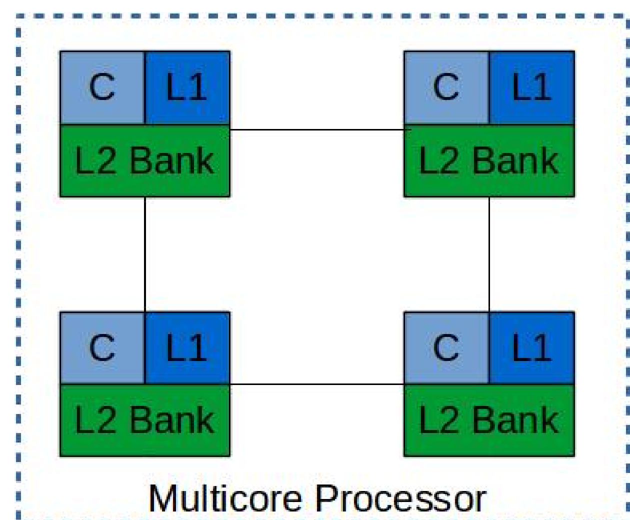

Fig. 1. An Example of Multicore Processor having 4 cores and 4 LLC banks.

*corresponding author

- **Eviction:** During the replacement process which block should be evicted from the cache. In case of LRU replacement policy, the eviction mechanism always selects the least recently used block as a victim block.
- **Insertion:** The newly incoming block replaces the victim block in the cache. But its position w.r.t. the replacement policy is determined by its insertion mechanism. In case of LRU replacement policy the newly incoming block is always placed in the MRU position.
- **Promotion:** This operation handles about what to do when a block is bing accessed from the cache. In case of LRU replacement policy, the promotion mechanism makes such blocks as MRU. It means that if a block B present in the bank and a request has been made to access the block. In this case the block will be accessed from the bank and hence it considered as hit. But after the access, the block will be moved to the MRU position.

The main reason to use such highly associative caches even in presence of hardware overheads is the performance. A highly associative cache reduces the conflict misses in the system and hence improves the system performance. These motivates researchers to reduces the hardware overhead of such highly associative caches. In this work we are mainly targeting the hardware overhead of the replacement policy of such cache.

The LRU based replacement policy are simple but faces a problem of dead blocks [2], [3], [4], [5], [6]. Since the insertion mechanism of LRU inserts a block at the MRU position and the eviction mechanism selects the victim from the LRU position, it takes long time in a highly associative cache to make a block LRU from MRU. It has been found that there are some blocks which are accessed only once in the cache. Such blocks though never used again but not possible to remove until they become LRU. Such block are called *dead-blocks*. Alternatively, a block is called dead-block at a particular instance, if the block will never used in the future before evicting it from the cache. The LRU based policy faces the issue of dead block and many technology has already been proposed to reduces the presence of such blocks [7]. Most of these dead block prediction policies are costly in terms of storage capacity as they require to maintain additional bits for prediction. Our proposed work is capable of early eviction of dead block with minimized hardware cost. Though the proposed policy is not as smart as [7] in terms of dead block prediction but managed to reduce hardware cost significantly. The proposed policy attempts to reduce hardware cost of LRU based techniques with high dead block prediction ability so that hit rate of the memory can be improved

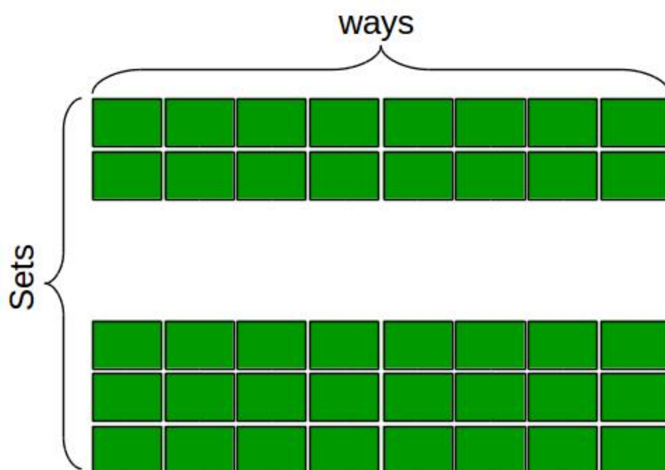

Fig. 2. An Example of 8-way Set Associative L2 Bank as shown in Fig. 1.

The organization of the paper is as follows. The next section discuss about the background and related works. The proposed Small-LRU is discussed in Section III. Section IV gives the experimental analysis and finally Section V concludes the paper.

II. Background

The simplest replacement policy is known as FIFO (Fisrt In First Out) which uses a straight forward strategy to replace victim block while the most widely used traditional replacement policy is LRU (Least Recently Used) which selects a victim block based on reference history. Some similar policies are MRU (Most Recently Used) and Random. In case of N-ways set-associative cache, LRU policy require $N \times log_2 N$ bits to represent a set. For example, a 4-ways set-associative cache require $log_2 4 = 2$bits to represent a single way and $4 \times 2 = 8$ bits require to represent a complete set. most of these traditional policies require the same hardware cost to implement. Random policy does not require any additional bit to select victim block but not suitable for general purpose.

Replacement policy is major area of research from the last two decades. The work in replacement policy can be divided into the following two categories: (a) Performance oriented like [7], [8], [9], [10] (b) Overhead reduction oriented like [11], [12]. The most efficient replacement policy was proposed in 1965 [13], which states that the evicted block must the block which will reuse in the furthest future. The policy is considered as the optimal replacement policy. Unfortunately it is not possible to implement this optimal policy in any physical computer as it needs the knowledge of future. Hence all the practical replacement policy proposed are trying to come closer to this policy in terms of performance. Note that, in case of replacement policy, the performance means reduction in the number of cache misses. There is a huge gap still exist between all the implementable replacement policies and the Optimal replacement policy [7], [11].

Many recent replacement policies have attempted to mimic the functionality of the optimal replacement policy using the modern techniques like Machine Learning and Artificial Intelligence [7], [10]. Even after all such attempts the gap is still exists. In [10] the authors have introduced a replacement policy which can predict future based on its past experiences. Another similar technique has been proposed in [14].

Removing dead blocks from the cache is also a major responsibility of the replacement policies. The LRU based replacement policies fail to remove dead blocks early [7]. Since detecting dead block also needs the knowledge of future it can only be predicted with some efficient prediction mechanism. Some well known dead block prediction based replacement

policies are [7], [8], [15], [16]. Most of these techniques has high hardware overhead as they need to maintain some prediction tables. Some low-overhead based dead block predictors are [11], [12]. The technique proposed in this paper is also a low-overhead based replacement policy.

III. Small-LRU

In this work a highly associative set is divided into two parts: LRU-Part and Random-Part. The LRU-Part maintains LRU replacement policy while the Random-Part maintains random replacement policy. LRU-Part is relatively small and have only 30% of the total ways. Thus the technique is called Small-LRU. During the eviction of a block, the Random-Part selects a victim block randomly and place it to the LRU-part. To accommodate the block coming from Random-Part, LRU-Part removes its least recently used block. An example of Small-LRU is shown in Fig. 3.

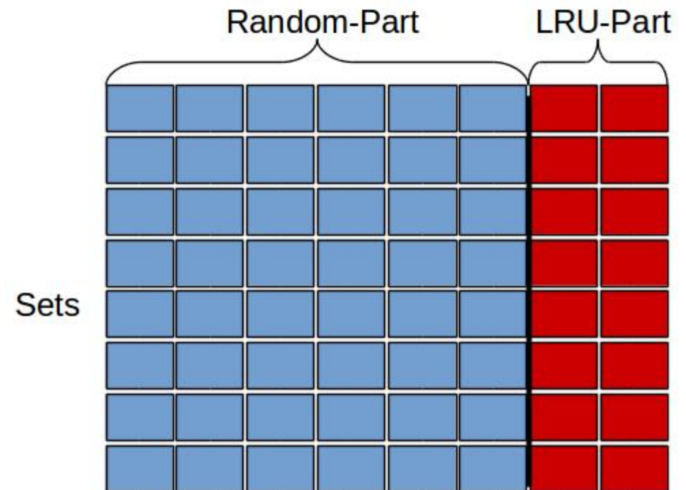

Fig. 3. An Example of Small-LRU, Implemented in a 8-way Set-Associative Cache.

The main advantage of Small-LRU can be divided into two parts: (a) Reducing hardware overhead and (b) Reducing the presence of dead blocks in the cache. The hardware overhead is largely reduced as the random replacement policy needs almost zero overhead. The dead blocks are present in any large sized cache memories. Since 70% of the cache is random the dead blocks will be evicted early from the set. On the other hand, if there is a highly used block being randomly picked by the random-part then block will be given chances by placing it in the MRU position of the LRU-Part. Another hit to the block will again move the block into the Random-Part.

The highly associative sets are used to remove the conflict misses in the cache. The proposed design is still equally capable of removing the conflict misses but with significantly reduced hardware overheads. Experimental analysis as discussed in Section IV shows that the proposed mechanism is good for most of the benchmarks.

A. Replacement Operations

Insertion Policy: Every time a newly inserted block will be placed in the Random-Part. The insertion policy needs to move a block from Random-Part to the LRU-Part for making room for the incoming block. To place the migrated block from Random-Part to the LRU-Part, the LRU-Part removes its LRU block.

Promotion Policy: If block from LRU-Part is need to be promoted then it will be placed on the random-part. Promoting a block from LRU-Part to Random-Part also needs some adjustments. Before doing such promotion a random block from the random-part will be moved to the LRU position of the LRU-Part.

Eviction Policy: Every time the evicted block is the LRU block of the LRU-Part.

B. Advantage of Small-LRU

1) Hardware Cost: As mentined in Section II, LRU policy requires to maintain $N \times log_2 N$ bits to represent each set of an N-ways set-associative cache. Since Small-LRU use random replacement policy on 70% ways and LRU only on 30% ways, the hardware overhead is reduces by more than 70%. In Small-LRU the number of additional bits required is $M \times log_2 M$, where $M = 0.3 \times N$. Table I represents the improvement achived in terms of hardware cost. It is observed that Small-LRU policy have reduced storage cost of LRU policy by 77% to 91% depending on the degree of associativity.

2) Dead Block Prediction: Early prediction of dead block is always a challenging task as discussed in Section I. Removing a dead block early from the cache is always a better choice. In Small-LRU, R-Part randomly moves a block to the MRU position of LRU-Part. The LRU-Part has only 30% ways from the total ways available in the cache. Hence even the original cache is highly associative a dead block can be removed early from the cache. A smarter replacement policy like DIP [9] or RRIP [15] in the LRU-Part may help to remove dead blocks even better.

TABLE I. Comparison of the bits required to implement original LRU policy and the proposed Small-LRU.

Associativity	Bits Required in Original LRU	Bits Required in Small LRU	Reduction in Small-LRU
8	24	2	91%
16	64	8	87%
32	160	29	82%
64	384	81	78%
128	896	199	77%

TABLE II. SPECIFICATIONS USED FOR DESIGNING SMALL-LRU.

Specification	Values
Cores used	4
Levels used in cache	2
Private cache	L1
Shared cache	L2 (total 4 banks)
L2 cache	512KB (per bank), 8-way set associative
L1 cache	64KB, 2-way set associative
Size of cache-block	64B

IV. EXPERIMENTAL ANALYSIS

System Architecture is implemented in gem5, a full-system simulator [17]. We have simulated a multicore processor (4 cores) with two level of cache memory. The upper level cache L1 is used as private cache to each core and last level cache LLC is shared among the cores. The baseline replacement policy as well as the proposed policies are implemented using Ruby module. PARSEC benchmark [18] applications are simulated in ALPHA system architecture.

To analyze the performance, all the benchmark applications are executed on the target machine designed using proposed replacement policy as well as the baseline replacement policies 200 million cycles. the specification of target machine used to implement Small-LRU is shown in Table II.

A. Result Analysis with Baseline-1

We considered LRU policy as baseline-1 to compare the result of our proposed policy. Statistical comparison in terms of MPKI (Miss Per Kilo Instructions) is shown in Fig. 4. It is observed from the figure that that Small-LRU have reduced MPKI by removing dead block early from cache. Fig. 5 shows the improvement in CPI (Cycle Per Instructions) due to the reduction in the number of cache miss. The proposed policy reduces MPKI by 10% and CPI by 5% on average. Though the improvement in the performance of the system is not significant with Small-LRU policy but the major advantage of this policy is the reduction of hardware cost without suffering the performance of the system. Fig. 6 depicts the percentage of bits reduction to implement Small-LRU compared to baseline-1 which is between 77% to 91%.

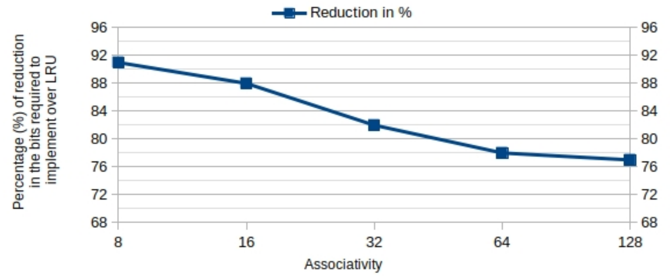

Fig. 6. The Percentage of Bits Reduction to Implement Small-LRU over Baseline-1.

B. Result Analysis with Baseline-2

We have also compared Small-LRU with a multicore processor having 16-way associative banks. We call this design as Baseline-2. Fig. 7 shows that Small-LRU reduces the MPKI by 10% in comparison to the MPKI of baseline-2. By comparing the CPI of both the techniques it is observed that Small-LRU improves the performance of the system by 5.5% in comparison to baseline-2. Fig. 8 shows the statistical

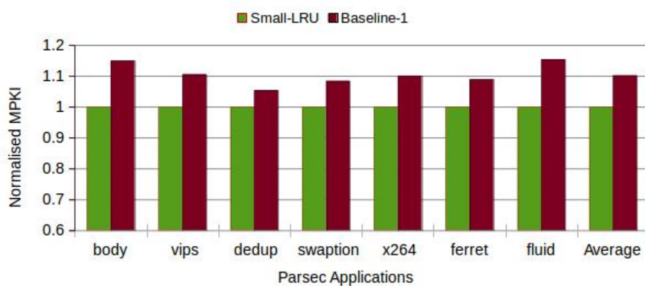

Fig. 4. Normalized Comparison of Small-LRU with Baseline-1 over MPKI.

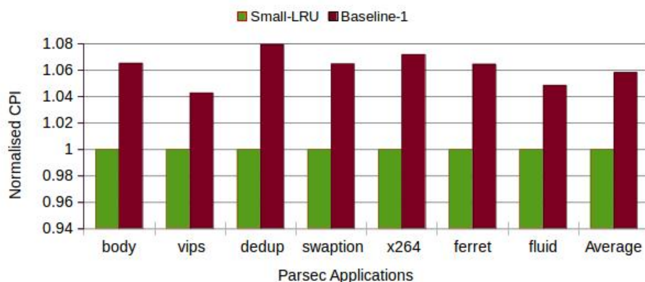

Fig. 5. Normalized Comparison of Small-LRU with Baseline-1 over CPI.

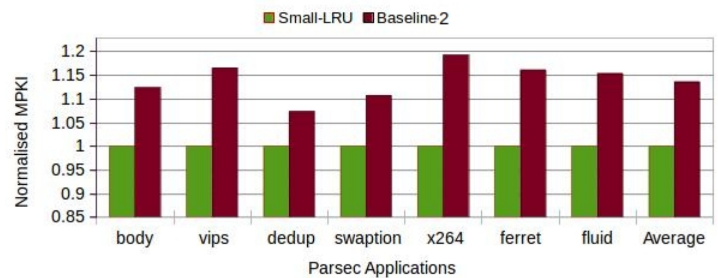

Fig. 7. Normalized Comparison of Small-LRU with Baseline-2 over MPKI.

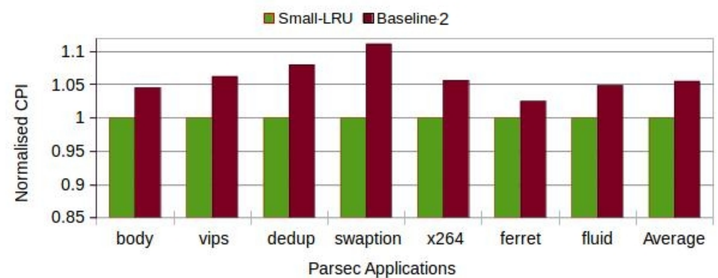

Fig. 8. Normalized Comparison of Small-LRU with Baseline-2 over CPI.

comparison of CPI between the Small-LRU and Baseline-2. Same as Baseline-1, the CPI improvement of Small-LRU over Baseline-2 is not significantly higher but enough to prove that Small-LRU reduces hardware overhead without degrading the performance.

V. Conclusion

Replacement policy plays a major role in improving the performance of the modern highly associative cache memories. As the demand of data intensive application is increasing it is highly required that the size of the Last Level Cache (LLC) must be increased. Increasing the size of the LLC also increases the associativity of the cache. Modern LLCs are divided into multiple banks where each bank is a set-associative cache. The replacement policy implemented on such highly associative banks consume significant hardware (storage and area) overhead. Also the Least Recently Used (LRU) based replacement policy has an issue of dead blocks. A block in the cache is called dead, if the block is not used in the future before its eviction from the cache. Removing such dead block early from the cache is not possible in LRU policy.

In this paper we have proposed Small-LRU policy to reduce the hardware cost by more than 70% and also improves the performance by removing early dead blocks. In this policy random replacement is used for 70% ways and LRU is applied for rest of the ways. A block is always inserted in the Random-Part and evicted from the LRU-Part. Early eviction of dead blocks improves the MPKI and CPI of system using Small-LRU by 10% and 5.5%, respectively.

References

[1] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: Association for Computing Machinery, 2002, p. 211–222.

[2] F. Juan and L. Chengyan, "An improved multi-core shared cache replacement algorithm," in *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*, Oct 2012, pp. 13–17.

[3] K. Morales and B. K. Lee, "Fixed segmented lru cache replacement scheme with selective caching," in *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, Dec 2012, pp. 199–200.

[4] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache replacement policies revisited: the case of p2p traffic," in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, April 2004, pp. 182–189.

[5] W. A. Wong and J. . Baer, "Modified lru policies for improving second-level cache behavior," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, Jan 2000, pp. 49–60.

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

[7] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, April 2008.

[8] Y. Xie and G. H. Loh, "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ISCA*, 2009.

[9] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. S. Emer, "Adaptive insertion policies for high performance caching," in *ISCA*, 2007.

[10] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 78–89.

[11] P. Das and B. R. Roy, "Splitways: An Efficient Replacement Policy for Larger Sized Cache Memory," *International Journal of Engineering and Advanced Technology (IJEAT)*, vol. 9, no. 1, 2019.

[12] S. Das, N. Polavarapu, P. D. Halwe, and H. K. Kapoor, "Random-lru: A replacement policy for chip multiprocessors," in *LSI Design and Test*, 2013, pp. 204–2013.

[13] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[14] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 110–123.

[15] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 60–71. [Online]. Available: https://doi.org/10.1145/1815961.1815971

[16] K. J. Deris and A. Baniasadi, "Analysis of non-optimal lru decisions in high-performance processors," in *2008 International Conference on Microelectronics*, Dec 2008, pp. 458–461.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011. [Online]. Available: http://parsec.cs.princeton.edu/