2011 24th Annual Conference on VLSI Design

LA-LRU: A Latency-Aware Replacement Policy for Variation Tolerant Caches

Aarul Jain
Cambridge Silicon Radio
Phoenix, AZ 85044, USA
aarul.jain@csr.com

Aviral Shrivastava
Arizona State University
Tempe, AZ 85287, USA
aviral.shrivastva@asu.edu

Chaitali Chakrabarti
Arizona State University
Tempe, AZ 85287, USA
chaitali@asu.edu

***Abstract*—Parameter variations in deep sub-micron integrated circuits cause chip characteristics to deviate during semiconductor fabrication process. These variations are dominant in memory systems such as caches and the delay spread due to process variation impacts the performance of a cache based system significantly. In this paper, we propose two schemes to reduce the performance impact of variations in caches: i) *Latency-Aware Least Recently Used (LA-LRU)* replacement policy which ensures that cache blocks that are affected by process variation are accessed less frequently, and ii) *Block Rearrangement* scheme that distributes cache blocks with high latencies to all sets uniformly. We implemented our schemes on the Wattch SimpleScalar toolset for Xscale, PowerPC and Alpha21264-like processor configurations. Our experiments on SPEC 2000 benchmarks show that our scheme improves the average memory access time of caches by 11% to 22%, almost eliminating any performance degradation due to variations. We also synthesized the *LA-LRU* logic, to find out that we can obtain this benefit at negligible increase in the power consumption of the cache.**

I. Introduction

The insatiable need of higher-performance in computing at all levels, from embedded to server, has been the main driving force behind technology scaling. After miniaturizing the transistor dimensions continuously for the past five decades, we have reached a point, where we are experiencing significant loss of control in the lithography as well as channel doping steps during manufacturing. Consequently, we observe large variations in the characteristics of manufactured devices. This phenomenon, called process variation has significant impact in terms of performance, power consumption, yield, and reliability of electronic designs [1], [2].

While no part of the processor is immune to variations, and to a large extent, process variations are random in nature, caches experience especially high amounts of variations. This is primarily because caches occupy majority of chip area, as well as transistors count. Both in embedded and high-end processors, caches comprise upwards of 90% of the transistor count and 60% of the area of the processor [3], [4].

Process variation in caches has several implications. First of all, due to process variations, some cache cell, or even an entire row/column may not work. If a cache cell is faulty, its error can be corrected by using error correcting codes [8]. However, to correct t errors, a minimum distance of $2t$ is required between the codewords which results in increase of area for storing large code words and impacts the decoding time severely. A faulty cache row/column can be replaced by a correctly working row/column [7]. Dynamically resizing the cache can fix errors in large parts of the cache [6]. Techniques like, way prioritization have been proposed to contain the extreme variations in the leakage power of the cache [9]. However, techniques that resize cache have performance penalties and may not be suitable for a wide variety of workloads.

Bennaser et al. [5] showed that due to process variations, cache access latency of a block may increase to two and even three clock cycles. The problem is that if different cache blocks have different latencies, then to operate correctly, maximum cache access latency of three clock cycles will be required to get the correct hit/miss information. However, such conservative approach of operating the whole cache at the latency of the slowest block may have very severe performance penalty. To reduce this penalty, [5] proposes a scheme to evaluate and store the latency of each cache block in a delay storage memory. A quick determination and lookup can then be made by reading the latency value of the block along with the tag and data.

This work builds upon adaptive cache architecture of [5] to further reduce the performance penalty due to process variations in the cache. This paper makes two independent, but additive contributions:

- Propose Latency-Aware LRU block replacement policy in caches, which tries to keep the frequently used data in the low latency blocks.
- Propose Block Rearrangement scheme to efficiently align similar latency blocks in the same way.

We have implemented our proposed techniques in the Wattch Simplescalar toolset [11], [12] for 32-way (XScale), 8-way (PowerPC) and 2-way (Alpha 21264) set associative caches. Our experimental results on benchmarks from SPEC 2000 shows that our scheme achieves 11% to 22% performance improvement over the adaptive architecture for various system architectures, *viz.*, XScale, PowerPC and Alpha21264, and is able to recover almost all the performance degradation due to process variations. We also synthesized the control logic for our scheme and find that this gain is at negligible power overhead.

The rest of the paper is organized as follows. In the next section, Section II we describe previous approaches to tackle with the performance problem due to process variations. Sec-

1063-9667/11 $26.00 © 2011 IEEE
DOI 10.1109/VLSID.2011.24

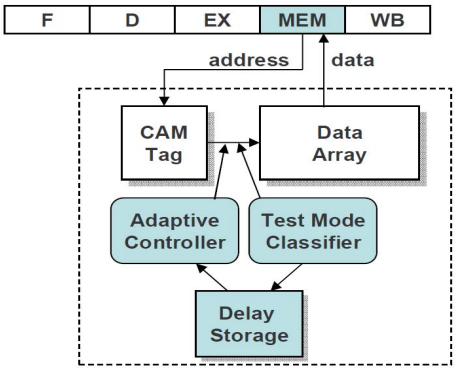

Fig. 1. Adaptive Cache Architecture: In the classification phase, the access latency of each phase is determined and stored in a 2-bit location per cache block. Our scheme, LA-LRU uses this latency value to improve performance.

tion III describes our approaches of Latency-Aware LRU, and Paired Block Rearrangement scheme. Section IV describes our experimental setup, results, and analysis, and finally Section V concludes the paper.

II. Relevant Work

A. Adaptive Cache Architecture to Tolerate Delay Variations in Caches

Since the location of the faults and the pattern of the variations is hard to determine before manufacturing, most process variation tolerant cache architectures require some scheme to detect the faulty or high-latency cache cells. The adaptive cache architecture [5] operates in two phases. During classification phase, done immediately after manufacturing, the BIST circuitry in the cache tests the entire cache and detects the speed of each cache block. The sense amplifiers sample the bitlines during the read operation in one, two and three cycles. The outputs of sense amplifier are compared with the original value and the actual cache latency for the cache blocks is determined and stored in a small delay storage memory array. During the execution phase, both the data array and delay storage units are accessed in parallel using row address of the index bits. The controller and the sense amplifiers are triggered at different time points depending upon the speed of access. It was found that the total area overhead for implementing this scheme is less than 1% of the total cache area and power consumption overhead was found to be less than 2% of the total cache power for a single word read.

B. Block Rearrangement to Reduce Accesses to High Latency Cache Blocks

In [10], a block rearrangement scheme was proposed that assumes that all blocks in a given set have the same latency which is decided by the worst case latency of all blocks in that set. The block rearrangement scheme modifies the address map of cache blocks to minimize the number of sets having high latency. They propose two schemes PairedBRT and PerfectBRT to change the set-to-block mapping, so that the blocks with the same latency are in the same set.

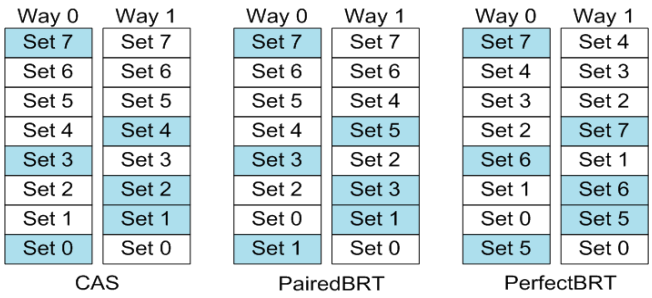

Fig. 2. Mutyam et al [10] propose PairedBRT and PerfectBRT, in which they attempt to aggregate similar latency blocks in the same way. In contrast, our techniques attempt to distribute high latency ways evenly among sets.

In the first technique, called PairedBRT, two adjacent sets are considered as a single group and block rearrangement can be performed within the group. In the second technique, PerfectBRT all sets within the cache form a group and block rearrangement can be performed between any two sets. For example, consider a 2-way associative cache shown in Figure 2, where the shaded blocks represent the long access latency blocks. The left sub-figure in Figure 2 shows the conventional block addressing scheme (CAS) in each way. In this case, the access latency of set 0 (comprised of block 0 in way 0 and block 0 of way 1) is long, since we have to wait for the hit response from the longer delay block before being sure about a hit or miss in the set. Similarly access latency of set 1 is also long. The middle sub-figure in Figure 2 shows that PairedBRT locally swaps the addressing of blocks 0 and 1. This makes the set 0 a fast set, even though set 1 remains slow. Comparing PairedBRT with conventional addressing scheme (CAS) it can be observed that CAS has six out of eight sets as high latency whereas PairedBRT has only four out of eight sets as high latency. The right sub-figure in Figure 2 shows that PerfectBRT, in which any two blocks can be rearranged, and therefore only 3 out of 8 sets are long latency. The performance of PerfectBRT is better than PairedBRT but the additional logic in address decoder affects the critical path delay of cache.

Both of these techniques, assume that the access latency of a set is the maximum of the access latencies of the blocks in the way. Consequently, they attempt to rearrange the cache blocks so that similar latency blocks are in the same way. In contrast, we build upon the adaptive cache architecture and use the latency information to rearrange high latency blocks evenly among the sets and improve performance by reducing the usage of high latency blocks.

III. Proposed Work

In this section we first describe the conventional LRU replacement scheme and then describe the Latency-Aware LRU (*LA-LRU*) scheme, which attempts to improve performance by keeping the most frequently used data in low-latency blocks. Then we describe our block rearrangement scheme that attempts to distribute the high latency blocks evenly among the sets. In combination with our LA-LRU, this scheme improves

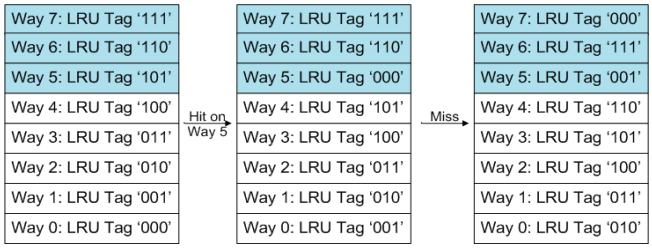

Fig. 3. Conventional LRU replacement scheme allows data to be mapped to long latency cache blocks.

performance by minimizing the use of high latency blocks.

A. Conventional LRU Replacement scheme

Figure 3 shows the conventional Least Recently Used Replacement mechanism for an eight way set associative cache. Each cache block in a set is assigned three tag bits which are updated on each cache access depending upon the cache line accessed. Here '000' represents the Most Recently Used way and '111' represents the Least Recently Used way. If there is a cache access and a hit occurs on way 5, the LRU tag of way 5 is updated to '000' and all LRU tags from '000' to '100' are incremented by one. If then there is a cache access and a miss occurs then the data in way 7 which has LRU tag '111' is the least recently used data and must be thrashed out. The new data fetched from main memory is then placed in way 7, its LRU tag is modified to '000' and all LRU tags from '000' to '110' are incremented by one.

Now consider a cache affected by process variation where say, three out of eight ways are high latency (shown as shaded in Figure 3). Temporal locality in workloads states that cache blocks accessed at a particular point of time will be accessed in near future. Thus Most Recently Used data in caches have higher probability of getting accessed in near future than the Least Recently Used data. In Figure 3 although initially least recently used data is in high latency ways but after a hit and a miss the most recently used data is present in high latency ways. Thus future cache access have more probability of having higher latency.

B. Latency-Aware LRU (LA-LRU) Replacement scheme

Figure 4 shows the Latency-Aware LRU replacement scheme. Here upon a hit on low latency ways the LRU tags are updated similar to the conventional LRU mechanism. Thus, if there is a hit on way 4, its LRU tag will be updated to '000' from '100' and LRU tags of way 0 to way 3 will be incremented by one. Now if there is a hit on high latency ways then the data in that way is moved to one of the low latency ways and the LRU data amongst low latency ways is moved to high latency ways. For example, in Figure 4, way 3 has the LRU data among low latency ways. Thus upon a hit on way 6 which is a high latency way, the data stored in way 3 and 6 are exchanged, the LRU tag of way 3 is updated to '000' and LRU tag of way 6 is updated to '101' as '101' is the tag corresponding to most recently used data among high latency ways. All the previous LRU tags from '000' to '101' are incremented by one. By exchanging data between high latency ways and low latency ways it can be ensured that most recently used data resides in low latency memory cells thereby ensuring that future access to this datum incur minimum penalty due to process variation.

Now consider the case where there is a miss in the cache. In this case, the LRU data which is present in high latency ways is thrashed out of the cache and LRU data is moved from low latency way to high latency way. The fetched data is then placed in the low latency way. For example, in Figure 4, upon a miss data is thrashed from way 7 which is the LRU datum, data from way 2 is moved to way 7 as way 2 has the LRU data amongst low latency ways and the fetched data is placed in way 2. The LRU tag of way 2 is updated to '000', LRU tag of way 7 is updated to '101' to indicate it is the most recently used data amongst high latency ways and other LRU tags incremented by one. Table I shows the average distribution of accesses to cache with LRU and *LA-LRU* based on latency required to access the data for SPEC 2000 using Wattch. It can be observed that number of single cycle access increases considerably thereby proving the effectiveness of the proposed replacement policy.

TABLE I
LA-LRU REDUCES THE PERCENTAGE OF ACCESSES THAT HAPPEN ON A LONG LATENCY CACHE BLOCKS ON SPEC 2000 (AVERAGE OVER ALL BENCHMARKS).

Type of access	% of cache accesses	
	LRU	LA-LRU
Hit with one cycle access	77.076	99.034
Hit with two cycle access	19.940	0.200
Hit with three cycle access	2.224	0.006
Miss	0.760	0.760

C. Implementation of LA-LRU

Figure 5 shows the block diagram of an implementation of process tolerant cache using *LA-LRU* mechanism. The block diagram is for a 64KB 8 way set associative cache with 32 bytes as block size. Here the delay storage unit provides information to Control Unit about the number of cycles it would take to access a particular block, and buffer 1 and buffer 2 are used to exchange data if there is miss, or a hit with access to a block with two cycle access latency or a hit with access to block with three cycle access latency. If the block to be accessed has one cycle access delay then the data is directly placed on the core bus. If the block requires two cycle access delay then it will require two extra cycles to update this two cycle latency block with the data from LRU block of latency one. However cache controller may stall the core for the third and fourth cycle only if there is another access pending to the cache. Similarly, if the block has three cycle access delay then buffer1 and buffer 2 are used to exchange data between LRU block with latency one and latency two respectively. Since writing the data to the three cycle latency block will take another 3 cycles, it takes overall 6 cycles to complete the cache transaction. However, the cache controller may stall the core for more than three cycles only if there is

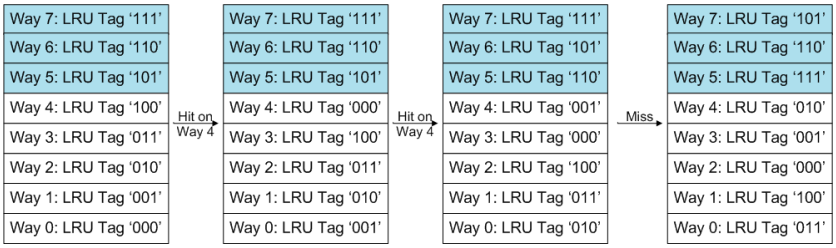

Fig. 4. On a hit, LA-LRU switches the data and the tag of the high latency block with a low latency block, so as to migrate the frequently accessed data to the low latency blocks.

another access to the cache within the 6 cycle window. Upon a miss, cache controller does not require to stall the core for write back operations since fetching the data from external memory/L2 cache takes more than 6 cycles.

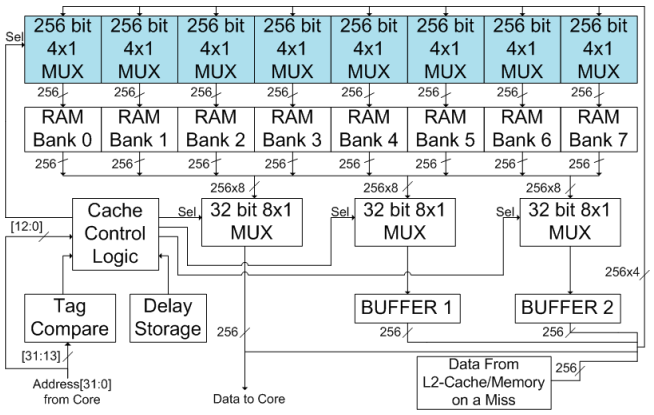

Fig. 5. Micro-architecture block diagram of process tolerant cache using *LA-LRU*

D. Block Rearrangement over LA-LRU Replacement scheme

We now describe techniques to modify and implement block re-arrangement proposed in [10] over *LA-LRU* to achieve even better performance. Since the distribution of high latency blocks is random, some of the sets in a cache will have more high latency ways as compared to other sets. We modify the address decoder and perform block rearrangement so that all sets have uniform distribution of high latency ways. By distributing the high latency ways among all the sets we ensure that all sets have at least few low latency ways from which data can be exchanged with high latency ways by *LA-LRU*. Thus, in PerfectBRT with *LA-LRU*, if there are m sets in k-way set associative cache which has t high latency blocks, then N_i the number of high latency ways in i^{th} set is given by,

$$N_i = \begin{cases} [t/m] + 1 & if\ i < (t)\ mod\ m; \\ [t/m] & if\ i \geq (t)\ mod\ m. \end{cases}$$

where $i\ \epsilon\ [0, 1, 2, ..., m-1]$.

The drawback of using perfect block rearrangement is the overhead of additional mux stages ($log_2 m$) so we propose to rearrange within a group of two sets so that only one additional mux stage is required in the address decoder of PairedBRT. The modified algorithm is as follows. If there are m sets in k-way set associative cache which has $t_{i,i+1}$ high latency blocks in i and $i+1$ set, then N_i the number of high latency ways in i^{th} set is given by,

$N_i = [t_{i,i+1}/2] + (t_{i,i+1})\ mod\ 2;$

$N_{i+1} = t_{i,i+1} - N_i.$

where $i\ \epsilon\ [0, 2, 4, ..., m-2]$,

As an example, in Figure 6 we show a 4 way set associative cache, where number of sets $m = 8$, associativity $k = 4$, and number of high latency blocks $t = 9$. In Figure 6(a) with no block rearrangement, $N_1, N_3 = 0;\ N_0, N_4, N_6, N_7 = 1;\ N_5 = 2\ and\ N_2 = 3$. In Figure 6(b) PerfectBRT rearranges the high latency blocks such that each set has uniform number of high latency ways, $N_0 = 2$ and $N_i = 1\ \forall\ i\ \epsilon\ [1, 2, ..., 7]$. In Figure 6(c) PairedBRT is shown where $t_{0,1} = 1$, $t_{2,3} = 3$, $t_{4,5} = 3$ and $t_{6,7} = 2$. After paired rearrangement, $N_1 = 0;\ N_0, N_3, N_5, N_6, N_7 = 1\ and\ N_2, N_4 = 2$.

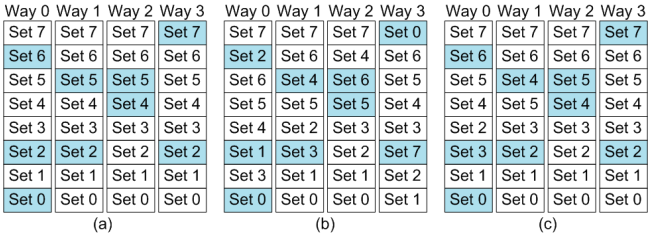

Fig. 6. (a) No block rearrangement, (b) Perfect block rearrangement for LA-LRU, (c) Paired block rearrangement for LA-LRU

IV. Simulation And Results

A. Simulation setup

We use Wattch Simplescalar architecture simulator [11] to benchmark performance and modify it to implement and compare *LA-LRU* and other schemes suggested in [5] and [10]. We simulated Xscale, PowerPC and Alpha21265-like processor configurations. The cache parameters for these systems are specified in Table II.

We use MATLAB to generate random distribution of delays and then use this distribution as base latency model in Wattch to deduce access latencies for each cache access. We use SPEC 2000 (cc1, equake, gzip, mcf, vortex and vpr) to benchmark the performance using Wattch. We fast forward 100 million

TABLE II
WATTCH SIMPLESCALAR CACHE PARAMETERS

	Xscale	PowerPC	Alpha21264
L1 Instruction Cache			
Size/Assoc./Block Size	32K/32/32	32K/8/32	64K/2/64
Access Latency	1 cycle	1 cycle	1 cycle
L1 Data Cache			
Size/Assoc./Block Size	32K/32/32	32K/8/32	64K/2/64
Access Latency	1-3 cycle	1-3 cycle	1-3 cycle
L2 Unified Cache			
Size/Assoc./Block Size	None	256K/2/64	512K/4/64
Access Latency	None	15 cycle	12 cycle
External Memory	First/Successive accesses = 160/24 cycles		

instructions and provide simulation results for next 300 million instructions.

We compare the following cache architectures:

1) **NPV** Cache with no process variation. All the access to the cache can be completed in a single cycle. Performance obtained in this scenario is the best possible performance that can be obtained.
2) **WORST** Assume that all cache blocks have access latency of three cycles.
3) **ADAPT** Adaptive cache architecture. The access latency depends on the access latency of the block containing the data [5].
4) **A-PairedBRT** Cache with paired block rearrangement technique, in which two adjacent blocks can be switched to aggregate similar latency blocks in sets [10].
5) **A-PerfectBRT** Cache with perfect block rearrangement technique, in which any two blocks can be switched to aggregate similar latency blocks in sets [10].
6) **LA-LRU** Our proposed block replacement technique in which we attempt to keep the most frequently used data in the low-latency blocks.
7) **LA-LRU + D-PairedBRT** LA-LRU with paired block rearrangement, in which two adjacent blocks can be switched to distribute high latency latency blocks evenly among sets.
8) **LA-LRU + D-PerfectBRT** LA-LRU with perfect block rearrangement, in which any two blocks can be switched to distribute high latency latency blocks evenly among sets.

We also vary the process variation model generated by MATLAB and show average performance degradation for each cache architecture for the following latency models [5]:

(a) 15% two cycle and 0% three cycle latency blocks.
(b) 25% two cycle and 1% three cycle latency blocks.

B. Performance for XScale, PowerPC and Alpha21264

We simulated Xscale, PowerPC and Alpha21264 like architectures and calculated average memory access using the following equation:

$AverageMemoryAccessTime = \sum_{i=1}^{3} Hit\_rate_i * i + Miss\_rate * Memory\_access\_latency$

where Hit_rate_i is the Hit ratio of blocks with latency i in L1 Data Cache.

Figure 7 shows the average memory access time for PowerPC using variation model (b). For all benchmarks simulated, we observe that *LA-LRU* gives almost same performance as the cache with no process variation. In Table III, we show the memory access time degradation for various cache architectures averaged over SPEC2000 benchmarks for XScale, PowerPC and Alpha21264.

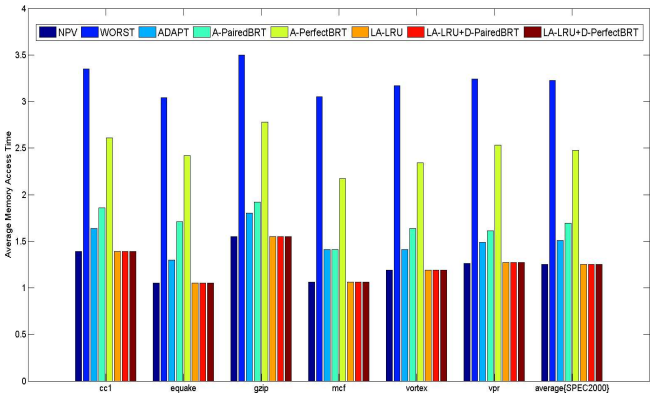

Fig. 7. Average memory access time for various cache architectures on Wattch for PowerPC using variation model (b)

The NPV (No Process Variation) cache architecture has theoretically the best possible IPC that can be achieved by any scheme designed for process tolerance and hence it is taken as the reference for comparing other cache architectures. The WORST cache architecture has the worst case performance with a performance degradation of almost 162% for XScale, 159% for PowerPC and about 173% for Alpha21264. For all three architectures (XScale, PowerPC and Alpha21264), A-PairedBRT has worse performance than the ADAPT cache architecture because there is limited gain due to rearrangement when the data required by an application is mapped to a high latency set of the cache. However, performance of ADAPT degrades as the number of high latency blocks increases. *LA-LRU* gives better performance that ADAPT because it ensures that the high latency blocks are accessed less frequently.

LA-LRU cache architecture has higher performance for high associative caches since it ensures that high latency data always resides in Least recently used way. This is why *LA-LRU* does better for XScale and PowerPC compared to Alpha21264 which has a 2-way set associative cache. For low associativity, *LA-LRU* scheme has fewer number of blocks to move data if there is hit on high latency block, resulting in reduced performance. However, this reduction in performance can be compensated by using block rearrangement.

Table III also shows that for Xscale and PowerPC, *LA-LRU* with D-perfectBRT and D-PairedBRT give minimal performance gain over *LA-LRU*. Thus for such architectures, use of rearrangement with *LA-LRU* is not recommended. However, architectures which have low-associative caches such as Alpha21264, rearrangement can give almost same performance

TABLE III
AVERAGE DEGRADATION IN MEMORY ACCESS LATENCY FOR DIFFERENT LATENCY MODELS DESCRIBED IN SECTION IV-A

Cache Architecture	% degradation for XScale		% degradation for PowerPC		% degradation for Alpha21264	
	(a)	(b)	(a)	(b)	(a)	(b)
NPV	0.00	0.00	0.00	0.00	0.00	0.00
WORST	161.86	161.86	158.94	158.94	172.57	172.57
ADAPT	11.41	21.62	14.23	20.42	12.53	21.31
A-PairedBRT	61.53	100.21	45.56	64.24	23.47	36.98
A-PerfectBRT	28.84	36.70	19.83	23.00	3.96	8.13
LA-LRU	0.11	0.21	0.11	0.23	3.69	7.53
LA-LRU + D-PairedBRT	0.11	0.21	0.11	0.23	0.81	2.69
LA-LRU + D-PerfectBRT	0.10	0.20	0.10	0.20	0.19	0.36

as the NPV architecture thereby eliminating any performance degradation due to process variation.

C. Overhead

To evaluate the overhead of *LA-LRU*, we used the trace from *gzip* benchmark as an input to the gate level netlist as it was observed that *gzip* benchmark has the highest percentage of two cycle/three cycle accesses to cache as compared to other benchmarks. Thus simulating with *gzip* benchmarks provides the worst case power overhead. It was observed that the power overhead of this scheme is just 3.5% of the total LRU logic. This is primarily because the extra buffers can be clock gated since they are required only if there is an access to high latency ways and the percentage of such accesses is less than 1% as shown in Table I. From Wattch simulator the average power consumption of L1 data cache is 4-7% of the entire system power. Since power consumption of LRU logic is small compared to that of data cache, the power overhead of implementing this scheme is negligible.

D-perfectBRT and D-pairedBRT are used for low associative caches therefore, delay of additional mux stages to implement them will have negligible overhead on the large delay of the address decoder. Also, with *LA-LRU* there is overhead of additional cycles required to swap data between cache blocks upon a hit on high latency blocks. Since such accesses are reduced to less than 1% from almost 23% (Table I), the overhead of implementing *LA-LRU* with D-PairedBRT and D-PerfectBRT is insignificant.

D. Other replacement schemes

The basic working principle of *LA-LRU* scheme is to ensure the most recently used data remains in low latency ways. We implemented this technique for FIFO replacement policy in Wattch for 64Kb 8-way set associative L1 Data Cache. We observe similar performance gain, as much as 13% for variation model (a) and 22% for variation model (b). Thus this technique is effective for other replacement policies as well.

V. CONCLUSION

In this paper we proposed a new replacement scheme, called *LA-LRU*, which improves the performance of cache systems affected by process variation. We used the technique discussed in [5] to classify the cache blocks based on their latencies and then used a replacement policy that ensures that most recently used datums are in low latency blocks. We observed that *LA-LRU* gives almost 160% average performance gain for SPEC 2000 benchmark as compared to WORST cache architecture. We then implemented block re-arrangement to further improve performance. However, block rearrangement gives marginal performance gain over *LA-LRU* without block rearrangement for caches with high associativity but can improve the performance of a system with low associativity caches significantly. We implemented the LRU logic in HDL and showed that overhead of power consumption by using *LA-LRU* is negligible. Finally, we extended the scheme to replacement policies like FIFO and showed the effectiveness of this technique.

REFERENCES

[1] K. Bowman, et al., Impact of die-to-die and within die parameter fluctuations on the maximum clock frequency distribution for gigascale integration, IEEE J. of Solid-State Circuits, Feb. 2002, 37(2), pp. 183–190.

[2] S. Borkar, et al., Parameter Variations and Impact on Circuits and Microarchitecture, Proc. Design Automation Conf. (DAC 03), IEEE Press, 2003, pp. 338-342.

[3] S. Rusu, H. Muljono, and B. Cherkauer, "Itanium 2 Processor 6M: Higher Frequency and Larger L3 Cache," IEEE Micro, 2004, 24(2) pp. 10-18.

[4] L. T. Clark, E. J. Hoffman, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. "An Embedded 32-B Microprocessor Core for Low-Power and High-Performance Applications," IEEE Journal of Solid State Circuits, 36(11):1599-1608, 2001.

[5] M. Bennaser, et al., "Data Memory Subsystem Resilient to Process Variations," IEEE Transactions on VLSI Systems, Dec. 2008, 16(2), pp. 1631–1638.

[6] A. Agarwal, et al., "A process-tolerant cache architecture for improved yield in nanoscale technologies", IEEE Transactions on VLSI Systems, Jan. 2005, 13(1), pp. 27–38.

[7] S. E. Schuster, Multiple word/bit line redundancy for semiconductor memories, IEEE J. Solid-State Circuits, Oct. 1978, 13(5), pp. 698-703.

[8] H. L. Kalter, et al., A 50-ns 16-Mb DRAM with a 10 ns data rate and on-chip ECC, IEEE J. Solid-State Circuits, Oct. 1990, 25(5), pp. 1118-1128.

[9] K. Meng, et al., "Process variation aware cache leakage management," Proceedings of the 2006 International Symposium on Low Power Electronics and Design, 2006, pp. 262–267.

[10] M. Mutyam, et al., "Working with Process Variation Aware Caches," Proceedings of the Conference on Design, Automation and Test in Europe, 2007, pp. 1152–1157.

[11] D. Burger, T. M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Tech. Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[12] D. Brooks, V. Tiwari, and M Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," Proceedings of the 27th annual international symposium on Computer architecture, 2000, pp. 83–94