Low-Latency Mechanisms for Near-Threshold Operation of Private Caches in Shared Memory Multicores

Farrukh Hijaz, Qingchuan Shi, Omer Khan
University of Connecticut, Storrs, CT, USA
{farrukh.hijaz,qingchuan.shi,khan}@uconn.edu

Abstract

Near-threshold voltage operation is widely acknowledged as a potential mechanism to achieve an order of magnitude reduction in energy consumption in future processors. However, processors cannot operate reliably below a minimum voltage, Vccmin, since hardware components may fail. SRAM bitcell failures in memory structures, such as caches, typically determine the Vccmin for a processor. Although the last-level shared caches (LLC) in modern multicores are protected using error correcting codes (ECC), the private caches have been left unprotected due to their performance sensitivity to the latency overhead of the ECC. This limits the operation of the processor at near-threshold voltages.

In this paper, we propose mechanisms for near-threshold operation of private caches that do not require ECC support. First, we present a fine-grain mechanism to disable cache lines in private caches, with bitcell failures at the target near-threshold voltage. Second, we propose two mechanisms to better manage the capacity-stressed private caches. (1) We utilize the OS-level data classification of private and shared data and evaluate a data placement mechanism that dynamically relocates the private data blocks to the LLC slice that is physically co-located with the requesting core. (2) We propose an in-hardware yet low-overhead runtime profiling of the locality of each cache line that is classified as private *data, and only allow such data to be cached in the private caches if it shows high spatio-temporal locality. These mechanisms allow the private caches to rely on the local LLC slice to cache the low-locality private data efficiently, and enable more space to hold the more frequently used private data (as well as the shared data). We show that combining cache line disabling with efficient cache management of private data performs better (in terms of application completion times) than using a single error correction double error detection (SECDED) based ECC mechanism and/or cache line disabling.*

1. Introduction

Although technology trends indicate a clear path to the integration of many cores on a single die, future multicore processors will be constrained by their energy efficiency [21]. Voltage scaling has emerged as an efficient mechanism to improve the energy efficiency of processors [9]. However, processors cannot operate reliably below a minimum voltage, Vccmin, since hardware components may fail. We measured (Figure 1) the energy consumption of standard 6T SRAM cells in a 16nm predictive technology node [11], and validated that near-threshold operation can result in more than 5× energy reduction. SRAMs specifically pose a critical limitation in low-voltage operating conditions because their functionality margins are lower. Standard 6T SRAM bitcells have greatly reduced read and write static-noise-margin (SNM) compared to standard CMOS logic gates. Therefore, as voltage is scaled, the read and write margins can easily be violated, particularly since SRAM cells exhibit extreme variation because of their use of minimally sized transistors and architectural requirements to maximize array size for area efficiency [9].

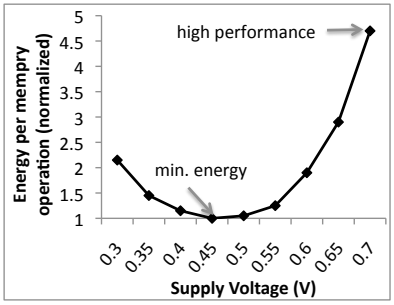

Figure 1: For a 16nm predictive technology, the dynamic voltage scaling system can scale its operating voltage from the nominal 0.7V down to a minimum energy point of 0.45V for the SRAM bitcells; a 5× energy advantage at 0.45V. However, the high bit error rates cause the SRAM bitcells to fail to operate at this near-threshold voltage.

Therefore, bitcell failures and the resilience mechanisms for SRAM memory structures, such as caches, typically determine the operating voltage for a processor.

Modern shared memory multicores [2, 7, 15] incorporate large logically shared but physically distributed last-level caches (LLC) to hold the working set of applications on chip, leading to non-uniform cache access (NUCA) [12]. The private caches, on the other hand, use cache coherence mechanisms to enable fast/local data access by allowing automatic data sharing and replication. Although the LLC has been traditionally protected using error correcting codes (ECC), the private caches (for example the single-cycle L1 caches) have been left unprotected due to their performance sensitivity to the additional latency of ECC [16]. This limits the operation of the processor at lower voltages. Because future processors will need to operate at near-threshold voltages for energy-efficiency, protecting private caches against bitcell failures is increasingly becoming critical.

In this paper, we propose mechanisms for near-threshold operation of private L1 caches that do not require ECC support, yet deliver low-latency on-chip memory access.

1. We present a fine-grain mechanism that can disable cache lines with one or more bitcell failures. We show that a practical ECC scheme (e.g., SECDED) that adds an extra cycle latency to the L1 caches, adversely effects performance. Additionally at near-threshold voltage, many cache lines are expected to incur more than 1-bit failures that cannot be handled using SECDED. Therefore a mechanism that can disable cache lines is required for the correct operation of the processor.
2. We show that a data placement mechanism that dynamically classifies and places private data in the LLC slice of the requesting core (similar to Reactive-NUCA [10]), allows the capacity-stressed private caches to rely on the relatively much larger LLC slice to hold the evicted data. Consequently, we show that migrating private data to the requesting core is critical to the overall performance of a multicore processor operating at near-threshold voltage, because many private cache misses that were previously served from remote LLC slices now become low-latency local LLC accesses.

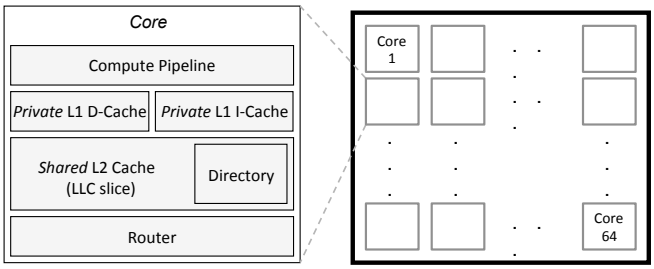

Figure 2: Baseline NoC-based 64-core Multicore. Each core implements a compute pipeline, a 2-level cache hierarchy and a router for communication. The L1 caches are organized as private to the core, while the L2 is logically shared but physically distributed across all cores.

3. We make an observation that in traditional processors the private caches are largely left unmanaged. A request for data allocates a data block in the private cache even if the data has no spatial or temporal locality. This leads to cache pollution since such low locality data can displace more frequently used data. We propose a novel mechanism for *locality-aware* caching of private data in the private caches. The key idea is to better manage the capacity-stressed caches by not allowing low-locality data to reside in the private caches. Instead, data blocks that exhibit low-locality are only allowed to be cached in the locally mapped LLC slice, giving the requesting core efficient access to such data. This extra cache capacity can now be used to hold more frequently used private and/or shared data. Our approach relies on runtime hardware-level profiling and tracking of the locality of each cache line that is classified as private data. Initially, when a core makes a memory request for private data, the cache line is brought into the private cache and a locality counter is subsequently updated. Upon an eviction from the private cache, the locality information is now tracked in the local LLC slice. If the locality of a data block is above a *locality threshold*, a miss from the private cache brings the entire cache line. Otherwise, the requested word at the LLC slice is accessed and the locality counter at the local LLC is updated for subsequent locality classification. Using this mechanism, we only allow private caching for data blocks with high spatio-temporal locality.

We simulate a mesh NoC-based 64-core multicore running multi-threaded applications. We show the benefits of the proposed mechanisms using a 2-level Private-L1, Shared-L2 (LLC) cache organization (Figure 2). The LLC is physically distributed in slices throughout the chip, and cache lines are statically and evenly interleaved across all cores. Our results show that using SECDED for single bit failures, and cache line disabling for multiple bit failures result in 1.45× performance loss compared to an ideal private cache that can operate without any bitcell failures at near-threshold voltage. Comparatively, using our mechanisms that combine cache line disabling with better cache management of private data results in <1.2× performance degradation compared to the ideal cache.

2. Mechanisms for Near-Threshold Operation of Private Caches

In this section we describe the various mechanisms for handling bitcell failures in the on-chip private caches.

2.1. Error Correcting Codes (ECC)

ECC provides protection against errors with a low overhead for the LLC, which can tolerate and/or hide higher latencies [22]. However,

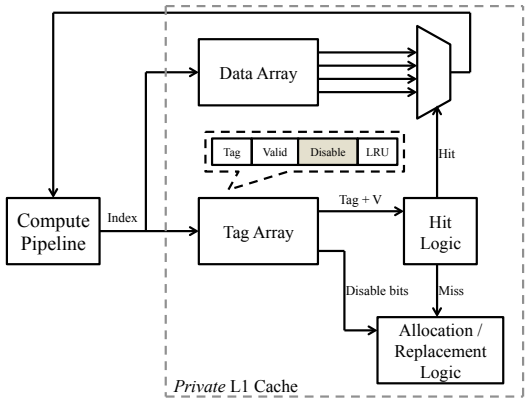

Figure 3: Private L1 caches add an extra *disable bit* for each line in the tag array. In a set associative cache the *disable bits* are used to select a functional way within a set. This allocation mechanism does not impact the hit latency of the cache. For the case where all the ways are disabled for a set, either the access can be treated as non-cacheable or an index remapping scheme can be used to allocate a cache line.

the effectiveness of ECC at L1 cache level is very limited, as the excessive latency results in severe performance loss. We observe that the basic single error correction scheme, such as SECDED that adds a single cycle to the hit latency of L1, on its own cannot provide enough protection against multi-bit errors. We conclude that we need to augment SECDED with some other scheme to make it work at near-threshold voltages.

One option is to increase the strength of ECC from SECDED to DECTED (or even stronger ECC), but this is not feasible as it will add additional latency to the L1 cache accesses. Another option is to disable part of the cache that has multi-bit errors. Previously, work has been done on selectively disabling portions of the cache [5, 19, 20]. All these techniques are coarse grain and end up disabling a substantial area of the cache that may still be functional. In way-disabling, the size of the cache decreases by 25% if we disable only one way in a 4-way cache. Set-disabling is even more severe, as an even greater portion of the cache needs to be disabled. In [20], the authors propose a hybrid way and set disabling scheme, but it is still coarse grain and results in wasted capacity.

2.2. Cache Line Disabling

Techniques that leverage disabling portions of a cache at sub-cache-line granularity have been studied before [17, 6]. Abella et al. [3] showed that fine-grain disabling at cache line and sub block works best in systems operating at low voltages with high error rate. Therefore, we decided to implement a fine-grain cache line level disabling mechanism, as shown in Figure 3. This scheme provides us with greater control on which portion of the cache to disable and results in higher available private cache capacity at near-threshold voltages.

We mark all faulty cache lines by setting the corresponding *disable bits* to "1". The disable bit is like any other tag bit and is read on every tag read. Once disabled, the cache line is not allocated by our modified LRU replacement scheme, as the allocation logic now chooses from the available cache lines in that set. The modified LRU logic is straightforward to implement and adds minimal amount of logic. The case in which all ways are disabled has a very low probability of occurrence and indeed we did not observe such scenario in our experiments. However, if that case does occur, a simple mechanism that replays the cache access on a remapped address can be introduced. We chose to simply treat such addresses as non-cacheable in the private caches. It is important to note that our

implementation does not add additional latency to the L1 hit logic, as shown in Figure 3.

Our fine-grain cache line disabling mechanism provides a way to operate at near-threshold voltages. However, the available capacity is decreased significantly at such aggressive voltages. At this reduced capacity, the miss rate of private caches shoots up steeply, resulting in severe performance degradation. Therefore, it is important to better manage the capacity-stressed private caches, such that the performance penalty is minimized.

3. Managing Capacity-Stressed Private Caches

3.1. Intelligent Data Placement

The *homing* of data blocks in the shared LLC becomes a critical factor when the capacity of private caches is constrained. The private cache miss latency not only depends on the access latency of the LLC but also its physical location on the chip. If an LLC slice is not physically co-located with the private cache, each such request has to complete an additional round-trip over the interconnection network. This network latency increases as more cores are added since the diameter of most on-chip networks increases with the number of cores. Therefore, it is desirable to *home* data blocks in the LLC such that their distance from the requesting core is minimized.

We propose to classify data as *private* or *shared* using the Reactive-NUCA's operating system based data classification mechanism [10]. Furthermore, Reactive-NUCA places private data at the LLC slice of the requesting core, interleaves shared data at the OS page granularity across all LLC slices, and replicates instructions at a single LLC slice for every cluster of 4 cores using a rotational interleaving mechanism. The *homing* of private data close to the requesting core has two major advantages: (1) many private cache misses that were previously served from remote LLC slices now become low-latency local LLC accesses, and (2) the capacity-stressed private caches can now rely on the relatively much larger LLC slice to hold the evicted data, thereby improving data locality.

On the other hand, this scheme suffers from a major drawback i.e., it leaves the private caches unmanaged. A request allocates and replicates a data block in the private caches even if the data has no spatial or temporal locality. This leads to cache pollution since such low locality data can displace more frequently used data. Therefore, in capacity-stressed private caches, paying attention to the locality of data becomes an important criterion for improving performance.

3.2. Locality-Aware Private Caches

We propose a *locality-aware* mechanism (similar to [13]) for private caches that ensures an L1 cache gets a private copy of a cache line from its local LLC slice only if it has high spatio-temporal locality. Otherwise, for low-locality data, a *word* is accessed in the local LLC slice. By selectively allocating cache lines, our mechanism prevents the pollution of L1 caches with low locality data and makes better use of their capacity.

Protocol Operation: We first define a few terms to facilitate describing our protocol.

- **Locality Counter**: Locality of a cache line is the number of times it is used (read or written) by a core in its private caches before it gets evicted. Our mechanism only tracks locality for private data that is co-located with the LLC slice of the requesting core.
- **Locality Threshold (LT)**: The cache line locality for which an L1 cache is granted a private copy of a cache line.
- **Cache-Line Accessible**: A cache line is "cache line accessible" by the L1 cache if its locality is $\geq LT$.
- **Word Accessible**: A cache line is "word accessible" by the L1 cache if its locality is $< LT$.

Let us understand how our mechanism works by considering its handling of read and write requests.

Consider read requests first. When a core makes a read request for private data and misses in its private L1 cache, the request is sent to the local LLC slice. The LLC hands out a private copy of the cache line if it is marked as *cache line accessible* in its integrated directory. The private cache then tracks the locality of the cache line by incrementing a locality counter for every subsequent read. *Locality counter* bits are added to each cache line in the private L1 cache's tag arrays, as shown in Figure 4. When the cache line is removed from the private cache due to eviction (conflict or capacity miss), the locality counter is communicated to the directory (integrated in the L2 tag array) with the acknowledgement message. The directory uses this information to determine if the cache line should be a marked as *cache line or word accessible* by comparing this locality counter with the LT.

On the other hand, if the cache line is marked as *word accessible*, the LLC replies with the requested word. If the cache line is not present in the LLC, it is brought in from off-chip memory. The directory also increments the locality counter (shown in Figure 4; at start-up, the locality counter in the directory is set to '0'). If the locality counter has reached LT, the cache line is marked as *cache line accessible* and a copy of the cache line is handed over to the L1 cache.

Now consider write requests. When a core makes a write request for private data that misses in its L1 cache, the request is sent to the local LLC slice. If the directory is marked as *cache line accessible*, it hands out a private copy of the line to the L1 cache. The L1 cache tracks the locality of the cache line by updating the locality counter and sends this information to the directory when the line is removed (due to eviction). The directory uses this information to classify the cache line as *cache line or word accessible* for handling its future requests.

On the other hand, if the cache line is marked as a *word accessible*, the directory increments the locality counter for the cache line. If the locality counter has reached LT, the cache line is marked as *cache line accessible* and a private copy of the cache line is handed over to the L1 cache. Otherwise, the word is stored in the LLC slice.

It is important to note that an L1 miss to a remote LLC slice is always treated as *cache line accessible* when serviced.

Locality Tracking Storage Overhead: The *locality-aware* private caching mechanism described above implements a single utilization counter at the L1 as well as the LLC slice for each cache line. Although the exact size depends primarily on the LT value, we will assume 4 bits of storage for each locality counter (including a bit to track whether a line is *cache line or word accessible*). Therefore, adding a locality counter for each cache line (64 Bytes) in the private L1 and shared L2 caches adds only a small storage overhead of $<$ 1%.

Selection of *Locality Threshold*

The Locality Threshold (LT) is a parameter to our protocol and combining it with the observed spatio-temporal locality of cache lines, our protocol classifies data as *cache line or word accessible*. We empirically observed that capacity-stressed L1 caches show most improvements between LT values of 2 and 8. Beyond LT of 8, the

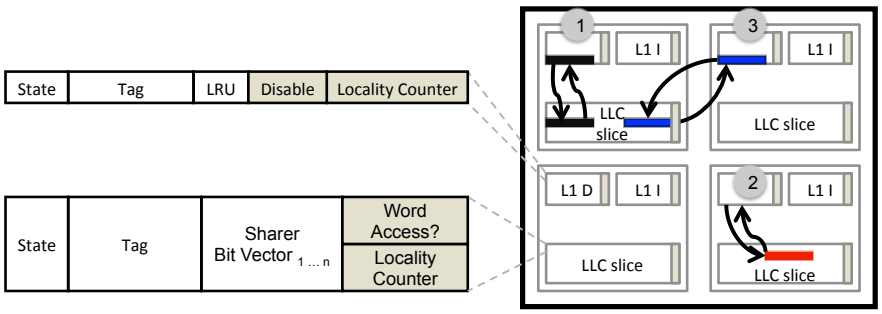

Figure 4: ①, ② and ③ are mockup requests showing the two modes of accessing on-chip LLC slices using our *locality-aware* protocol. Since the *black* data block has high locality with respect to the requesting private cache at ①, the LLC slice hands out a private copy of the cache line. On the other hand, the private low-locality *red data block* is always cached in a single location at the LLC slice, and all requests (②) are serviced as word accesses. ③ shows a shared data block and regardless of the locality, the LLC slice hands out a private copy of this cache line. We do not support locality tracking for shared data because it will require extra locality counters for each sharer in the system. Each entry in the private L1 cache's tag array is extended to include additional bits to track the locality of a cache line (top left). Each L2/directory entry is extended with a locality counter and a mode bit for the private L1 cache. The mode bit is used to differentiate a cache line access from a work access in the LLC slice.

locality of data is not exploited and the unnecessary word accesses start to hurt performance. Therefore, our choice of fixing the locality counter to 3 bits is sufficient.

Tradeoffs in Locality-Aware Private Caching: To analytically reason about the memory access latency under the *locality-aware* mechanism, we break down the memory accesses into fundamental components and present a first-order analytical model to estimate the average memory access latency (AML).

$$
\begin{aligned}
AML^{LT} = \\
& Rate_{\text{Private\_L1\_hit}} \times Cost_{\text{Private\_L1\_hit}} \\
+ & Rate_{\text{Local\_L2\_Cold\_miss}} \times Cost_{\text{Local\_L2\_Cold\_miss}} \\
+ & Rate_{\text{Local\_L2\_Capacity\_miss}} \times Cost_{\text{Local\_L2\_Capacity\_miss}} \\
+ & Rate_{\text{Local\_L2\_Sharing\_miss}} \times Cost_{\text{Local\_L2\_Sharing\_miss}} \\
+ & Rate_{\text{Local\_L2\_Word\_miss}} \times Cost_{\text{Local\_L2\_Remote\_miss}} \\
+ & Rate_{\text{Remote\_L2\_Cold\_miss}} \times Cost_{\text{Remote\_L2\_Cold\_miss}} \\
+ & Rate_{\text{Remote\_L2\_Capacity\_miss}} \times Cost_{\text{Remote\_L2\_Capacity\_miss}} \\
+ & Rate_{\text{Remote\_L2\_Sharing\_miss}} \times Cost_{\text{Remote\_L2\_Sharing\_miss}} \qquad (1)
\end{aligned}
$$

where, the miss rates constitute all memory accesses that miss the private L1 caches. While $Cost_{\text{Private\_L1\_hit}}$ mostly depends on the cache technology itself, we improve the latency of memory accesses by optimizing the other variables in Equation 1.

Our protocol intelligently utilizes the private cache capacity in each core. The private cache miss rate is strongly correlated to the working set size and the degree of sharing in an application. An application with a small working set that fits into the private caches may still benefit more from private caching even if most of its data has low spatio-temporal locality. On the other hand, an application whose working set stresses the capacity of the private caches will benefit from our mechanism, as it will selectively allow cache lines with high spatio-temporal locality to reside in the private caches and keep the low-locality private data pinned to the relatively much larger capacity local LLC slice.

This classification of cache lines as *cache line or word accessible* in our protocol depends on the observed locality counters as well as the LT value. Increasing the value of LT decreases the number of capacity and sharing misses (both local and remote) but potentially increases the number of word misses to the local LLC slice. The *number* and *cost* of such misses offers a tradeoff that affects the average memory access latency.

Architectural Parameter	Value
Number of Cores	64 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Physical Address Length	48 bits
Memory Subsystem	
L1-I Cache per core	32 KB, 4-way, Private, 1-cycle
L1-D Cache per core	32 KB, 4-way, Private, 1-cycle
L2 Cache per core	128 KB, 8-way Associative Shared, Inclusive, 8-cycles
Cache Line Size	64 bytes
Directory Storage per core	16 KB
Directory Protocol	Invalidation-based MESI Full-map
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	100 ns
Electrical 2-D Mesh with XY Routing	
Hop Latency	4 cycles (3-router, 1-link)
Router	Pipelined
Flit Width	64 bits
Header (Src, Dest, Addr, MsgType)	1 flit
Word Length	1 flit (64 bits)
Cache Line Length	8 flits

Table 1: Architectural parameters used for evaluation.

4. Evaluation Methodology

We evaluate a 64-core shared memory multicore using the Graphite simulator [14]. The important architectural parameters used for evaluation are shown in Table 1. We simulate thirteen SPLASH-2 [18] benchmarks, one PARSEC [8] benchmark, and a dynamic graph benchmark using the Graphite multicore simulator. The dynamic graph benchmark models a social networking application that finds connected components in a graph [1].

Each application is run to completion using the recommended input sets. For each simulation run, we measure the *Completion Time*, i.e., the time in parallel region of the benchmark; this includes the instruction processing latency, memory access latency, and the synchronization latency.

4.1. Near-threshold Operation of Private L1 Caches

We evaluate the following failure-tolerance mechanisms for the private L1 caches in the processor.

1. ***SECDED + Disable*** implements SECDED with an extra 1-cycle latency for the L1 caches. For cache lines with >1 bitcell failures, it uses the cache line disable mechanism discussed in Section 2.

2. ***Disable*** relies on the cache line disable mechanism to tolerate all bitcell failures. This mechanism does not add extra latency to the L1 cache as long as at least one of the ways is available for each set in the cache.
3. ***Disable + Data Placement***: In addition to cache line disable mechanism, data classification and migration is implemented for improved placement of private data in the LLC slice co-located with the requesting core (cf. Section 3.1).
4. ***Disable + Data Placement + Locality-Aware***: In addition to cache line disable as well as private data placement, the private L1 caches selectively allow data caching for cache lines that exhibit high spatio-temporal locality (cf. Section 3.2).

4.2. Cache Line *Disabling* for Private L1 Caches

In this paper, we use the Vccmin of 0.45V from Figure 1 as a reference near-threshold voltage. To generate the *disable bit mask* for each private L1 cache, which indicates how many errors are located in each cache line, we used the data points from Alameldeen's paper [4]. In figure 1 from [4], there are 4 lines that show the probabilities that a single cache line can have 1 to 4 errors. We captured the data points for 0.45V from the 4 lines in this figure, and used them to generate a *disable bit mask*.

Figure 5 shows our methodology to generate the *disable bit mask* for each private L1 cache. We first capture the error probabilities from [4], and generate a random variable for each cache line using standard uniform distribution on the open interval (0,1). The probability of 1 to 4 errors in a cache line is used as a threshold value at our near-threshold Vccmin of 0.45V. The probabilities are extracted as following: 1 error = 0.3926; 2 errors = 0.1802; 3 errors = 0.0827; 4 errors = 0.0325. If the random variable generated for a certain cache line is X $<$ 0.0325, then that cache line has 4 errors in the mask. If the random variable X $<$ 0.1152 and X $>$ 0.0325, then that cache line has 3 errors in the mask, and it works the same for 2 errors, 1 error, and 0 error. For each cache line, we repeat this process. The output of our *disable bit mask* at Vccmin = 0.45V has 33% of the cache lines in each private L1 cache with 0-bitcell failures, 43% with 1-bitcell failures, and 24% with $>$1-bitcell failures.

5. Results

In this section we evaluate the *completion time* tradeoffs with using the proposed mechanisms from Section 4.1. As shown in Figure 6, our results indicate that *Disable* is by far the worst performing system with an average of 1.5$\times$ performance degradation compared to an ideal system with 0-bitcell failures in the private L1 caches. Introducing *SECDED + Disable* allows many of the 1-bitcell failures to be corrected, therefore making an additional 43% cache lines available. However, this comes with the cost of an extra cycle penalty for each L1 cache hit. This results in some improvement over the *Disable* system, but *SECDED + Disable* still performs an average of 1.45$\times$ worse compared to the ideal system.

When the *private data placement* optimization is introduced along with cache disabling (*Disable + Data Placement*), many of the private cache misses to remote LLC slices now become lower latency misses to the local LLC slice. This results in an average of 20% improvement over the *SECDED + Disable* system and narrows the performance gap to an average of 1.25$\times$ worse when compared to the ideal system. Finally, when the locality-aware private caching mechanism is also considered (*Disable + Data Placement + Locality-Aware*), the capacity-stressed private caches perform within an average of 1.19$\times$

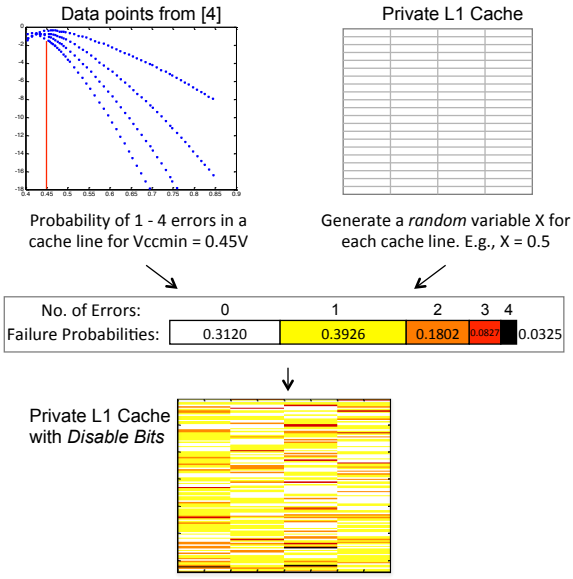

Figure 5: Based on the probability of a bitcell error, each cache line in the private L1 cache is marked with 0, 1, 2 ,3 or 4 bitcell failures.

of the ideal system. The benefits of the locality-aware mechanism are more prominent in benchmarks with higher number of private cache misses. For example, LU_NON_CONTIGUOUS incurred the highest private L1 cache miss rate of 12% among our benchmarks. Since locality-aware mechanism works best under capacity-stressed private caches (cf. Section 3.2), LU_NON_CONTIGUOUS improved performance by 25% over *Disable + Data Placement*. In general, we observe that for all other benchmarks, the locality aware mechanism always performs better or at par with the *Disable + Data Placement* system.

To further understand why locality-aware private caching outperforms other counterparts, we evaluate the private L1 cache miss ratios for the *Disable*, *Disable + Data Placement*, and *Disable + Data Placement + Locality-Aware* systems relative to the ideal system. Figure 7 shows the results for a subset of our benchmarks as others show similar trends. RADIX shows dramatic improvement by converting majority of the L1 misses into local LLC accesses. Furthermore, the locality-aware mechanism lowers the local (as well as remote) LLC capacity misses significantly and replaces these with relatively inexpensive word accesses. Although OCEAN_NON_CONTIGUOUS shows similar behavior as RADIX, LU_NON_CONTIGUOUS, in addition to significantly lowering capacity misses it also converts many of the local LLC sharing misses into lower latency word accesses. Finally, because the RAYTRACE benchmark has a low 3% private cache miss rate and most of the L1 misses result in accesses to remote LLC slices (RAYTRACE has significant shared data), the opportunities for performance improvement in this benchmark are limited.

6. Conclusion

In the imminent era of large-scale shared memory multicores, energy efficient operation will be critical. To enable operation at near-threshold voltages, such processors will need to handle the persistent errors due to process variations. We show in this paper that using ECC mechanisms at L1 cache level is not feasible due to its large overhead. We then propose a fine-grain cache line disabling mechanism to selectively disable faulty cache lines without wasting any cache capacity. Finally, we introduce two optimizations to better man-

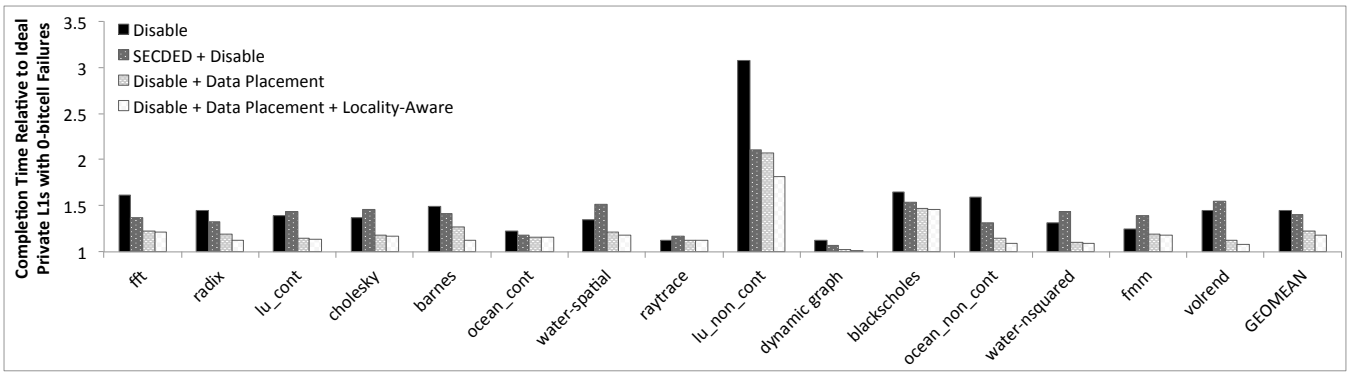

Figure 6: Completion time comparisons of capacity-stressed private caches operating at near-threshold voltage of 0.45V.

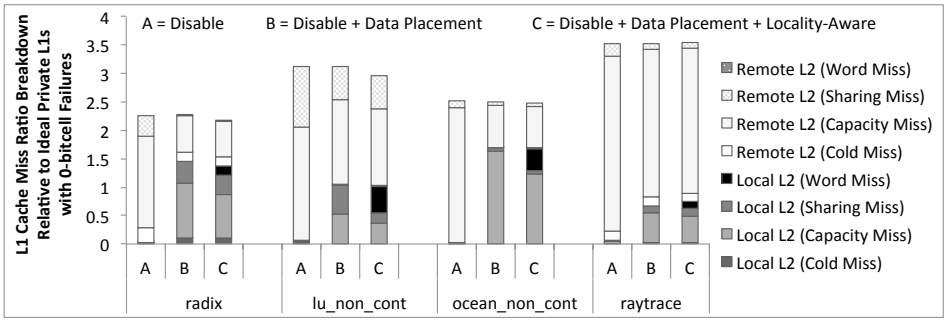

Figure 7: Variation of cache miss ratio breakdown. Private data placement allows many L1 misses to remote LLC slice to become low-latency misses to the local LLC slice. The locality-aware mechanism further lowers miss latency by converting more expensive sharing and capacity misses into word misses as well as make more room for useful data to reside in the private L1 caches.

age the capacity-stressed private caches. We show that combining cache line disabling with efficient cache management of private data performs within 20% of an ideal system (where the processor can operate at near-threshold voltages without incurring any hardware failures).

References

[1] "DARPA UHPC Program BAA," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-37/listing.html, March 2010.

[2] "Intel Xeon Processor E7-8867L," http://ark.intel.com/products/53577, 2011.

[3] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. González, "Low vccmin fault-tolerant cache with highly predictable performance," in *International Symposium on Microarchitecture*, 2009.

[4] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-efficient cache design using variable-strength error-correcting codes," in *International Symposium on Computer Architecture*, 2011.

[5] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *International Symposium on Microarchitecture*, 1999.

[6] A. Ansari, S. Feng, S. Gupta, and S. A. Mahlke, "Archipelago: A polymorphic cache design for enabling robust near-threshold operation," in *International Conference on High Performance Computer Architecture*, 2011.

[7] S. Bell *et al.*, "TILE64 - processor: A 64-Core SoC with mesh interconnect," in *International Solid-State Circuits Conference*, 2008.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[9] A. P. Chandrakasan, D. C. Daly, D. F. Finchelstein, J. Kwong, Y. K. Ramadass, M. E. Sinangil, V. Sze, and N. Verma, "Technologies for ultradynamic voltage scaling," *Proceedings of the IEEE*, vol. 98, pp. 191–214, 2009.

[10] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *International Symposium on Computer Architecture*, 2009.

[11] http://ptm.asu.edu, "Nanoscale integration and modeling group at asu."

[12] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *ASPLOS*, 2002.

[13] G. Kurian, O. Khan, and S. Devadas, "A Case for Fine-Grain Adaptive Cache Coherence," http://hdl.handle.net/1721.1/70909 MIT-CSAIL-TR-2012-012.

[14] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *International Conference on High Performance Computer Architecture*, 2010.

[15] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora, "A 45nm 8-core enterprise Xeon® processor," in *Asian Solid-State Circuits Conference*, 2009.

[16] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," in *IBM J. Res. Dev.*, vol. 46, no. 1, 2002, pp. 5–25.

[17] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *International Symposium on Computer Architecture*, 2008.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Conference on Computer Architecture*, 1995.

[19] S.-H. Yang, B. Falsafi, M. D. Powell, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches," in *International Symposium on High Performance Computer Architecture*, 2001.

[20] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *International Symposium on High Performance Computer Architecture*, 2002.

[21] K. Yelick, "Ten ways to waste a parallel computer," in *Keynote at International Symposium on Computer Architecture*, 2009.

[22] D. H. Yoon and M. Erez, "Memory mapped ecc: low-cost error protection for last level caches," in *International Symposium on Computer Architecture*, 2009.