Appears in the International Conference on Computer Design (ICCD)
San Jose, California, October, 2006

Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache

Nathan N. Sadler and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
{nns, sorin}@ee.duke.edu

Abstract-- **We deconstruct and compare the two dominant existing approaches for L1 data cache (L1D) error protection, with respect to performance, L2 cache bandwidth, power, and area. The two approaches are: (1) parity on the L1D with write-through to an ECC-protected L2, and (2) ECC protection on the L1D. Qualitatively, the first approach requires a write-through L1D, which places a large bandwidth and power demand on the L2. The second approach adds more bits in the L1D for error protection, which adds to the L1D's area and power while degrading its performance. Our quantitative results show that the relative costs of the second approach are small and that its benefits outweigh these costs. We also present a new error protection scheme, called the Punctured ECC Recovery Cache (PERC), that achieves the best features of both existing schemes.**

I. Introduction

Our primary goal in this paper is to evaluate the commercially dominant error correction schemes for L1 data caches (L1D), so that microarchitects will be able to choose the appropriate scheme for their system. We quantitatively compare these schemes with respect to four metrics: performance, L2 cache bandwidth, power consumption, and area. We do not evaluate the latency or power consumption of error recovery, since recovery is a rare event that has negligible impact. As with almost all other studies of fault tolerance for memory structures, our fault model includes single and multiple bit errors in cache blocks. We compare the fault tolerance of each scheme by how many bit errors it can *correct*; although being able to *detect* additional bit errors is helpful, it still requires OS intervention or a reboot.

In Section II., we discuss existing approaches for providing L1D error protection. Existing commercial microprocessors tend to use one of two approaches. The first option, used by the Pentium4 [9], UltraSPARC IV [15], and Power4 [5], among others, is to use an error detecting code (EDC) on the L1D. If an error is detected, then the data is recovered from the L2, in which every block is protected with an error correcting code (ECC). We refer to this scheme as *EDC/ECC*. The second option, used by the AMD K8 [1] and Alpha 21264 [11], is to use ECC on both the L1D and the L2. We call this scheme *ECC/ECC*. Qualitatively, the differences between EDC/ECC and ECC/ECC are that EDC/ECC leads to a smaller and faster L1D, at the expense of more L2 bandwidth and power.

In Section III., we present a new design, called the *Punctured ECC Recovery Cache (PERC),* that combines some of the best aspects of existing schemes. We add only the bits needed for error detection to the L1D, keeping it fast and small like EDC/ECC. If an error is detected, we then fetch the additional bits for error correction from the dedicated PERC. With PERC, we can use stronger ECC codes than with ECC/ECC while achieving the same performance as the more weakly protected ECC/ECC system,

Our experimental results, described in Section IV., use SimpleScalar [4] and CACTI [10] to compare the performance, L2 bandwidth usage, power, and die area of existing schemes and PERC. The results show that either ECC/ECC or PERC is generally preferable, based on all of these aspects.

The contributions of this paper are the following:

- Thorough experimental evaluation of existing schemes for L1D error protection,
- Quantitative results which demonstrate a clear advantage of ECC/ECC over EDC/ECC, and
- PERC, which tolerates more errors than ECC/ECC with slightly better performance and less power.

II. Existing Error Protection Schemes

Commercial microprocessors tend to use either EDC/ECC or ECC/ECC.

EDC/ECC. In all commercial systems we have found, the EDC is a simple parity bit that can detect single-bit errors, and the ECC can correct single bit errors. We will thus evaluate this "SEC-SED" (single error correction, single error detection) scheme in this paper. EDC/ECC requires a write-through L1D and cache inclusion. However, a write-through L1D significantly increases the bandwidth that is demanded of the L2 cache. The additional L2 accesses also drive up power consumption.

ECC/ECC. In most schemes we have found, the ECC can correct single bit errors and detect double and single bit errors; thus, we will evaluate ECC/ECC schemes with "SEC-DED" protection. ECC/ECC enables a write-back L1D and a non-inclusive cache hierarchy. However, ECC on the L1D also increases the size of each block more than EDC and thus degrades L1D access time. ECC can also increase access latency, with respect to EDC, due to its greater computational demands. In particular, a write to a subset of a block requires a read-modify-write (RMW) if the ECC is computed across the whole block. RMWs are slower than writes and may require

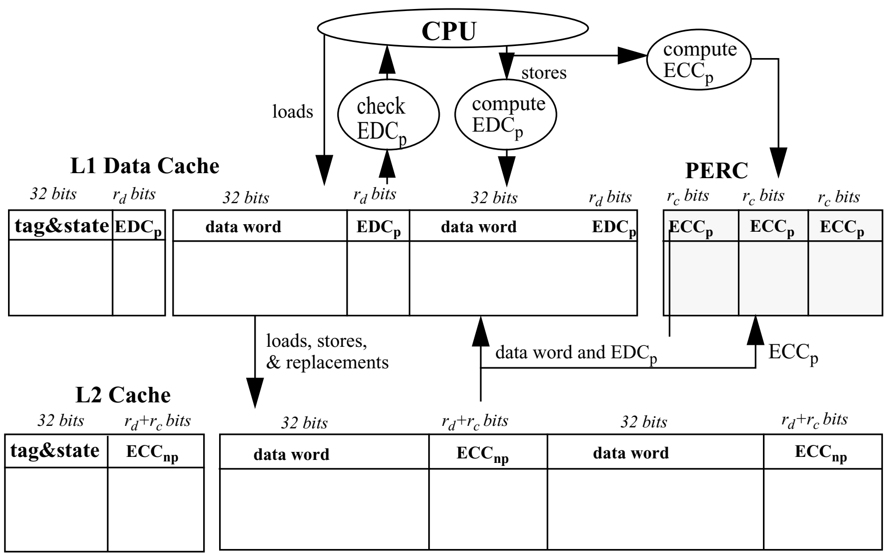


Fig. 1. Simplified example of cache hierarchy with PERC. We assume that block tags and state are 32 bits (28-bit tag, 4-bit state). We assume direct-mapped caches with data blocks that are only 8 bytes long (2 32-bit words). We do not show the recovery path for when an error is detected in the L1D and ECC must be fetched from PERC. The size of r_d and r_c depend on the particular punctured error code chosen. The example assumes that ECC_{np} is the concatenation of EDC_p and ECC_p.

adding ports on the cache. Applying ECC at the word granularity avoids RMWs, but it requires more check bits per block and thus a larger, slower, and more power-hungry L1D.

Other Schemes. Kim and Somani [12] use parity caching and shadow checking to selectively protect the most recently used cache blocks. Zhang et al. [17] maintain multiple copies of certain blocks in the L1D, and they use these replicas for error detection and correction. Several proposals [12, 13, 3] use early writebacks of dirty blocks (or scrubbing) to reduce the lifetime of dirty blocks in the L1D. Similarly, Asadi et al. [3] periodically refresh L1D blocks with data from the L2. None of these schemes can provide traditional guarantees for error coverage (i.e., all errors of type Y are tolerated), since coverage depends on data access patterns. Zhang [16] uses a replication cache (R-cache), which holds replicas outside the L1D to avoid using its precious resources just for replicas. The R-cache can protect all cache blocks, but entries in the small R-cache must frequently be written back to the L2, which places some bandwidth and power burden on the L2 (but less than EDC/ECC). Li et al. [13] protect clean L1D blocks with parity (and re-fetch from the ECC-protected L2 on an error) and dirty blocks with ECC.

III. Punctured ECC Recovery Cache

The goal of the PERC is to be able to tolerate errors in the L1D without significantly degrading performance, increasing L2 bandwidth, or increasing power usage.

A. *Punctured Error Codes*

Error codes add r check bits to each k-bit piece of data to create n-bit ($n=r+k$) codewords that contain information redundancy. The error detection and correction capabilities of a code are determined by its *Hamming distance*, which is the minimum number of bits in which any two codewords differ from each other. A code can detect d-bit errors with a Hamming distance of d+1 and correct c-bit errors with a distance of 2c+1.

For PERC, we leverage the properties of punctured error codes [6, 7]. What differentiates a punctured code is that the r check bits can be separated into r_d bits for detection and r_c bits for correction ($r = r_d + r_c$). We denote the r_d punctured error detection bits as EDC_p and the r_c punctured error correction bits as ECC_p, and we denote non-punctured codes with EDC_{np} and ECC_{np}. Given the datum and EDC_p, we can detect all d-bit errors. Given the datum, EDC_p, and ECC_p, we can correct all c-bit errors. The values of d and c depend on the Hamming distance of the chosen code.

In our PERC design, we use a punctured Reed-Solomon code. We assume that EDC_p and ECC_p will be added at the word granularity, instead of block granularity, to avoid having to perform RMWs for every store that does not write the entire block. The code adds 8 bits per 32-bit word. One bit provides a Hamming distance of two and is used for single-bit error detection, while the remaining seven bits provide single-bit error correction, which is the same as the EDC/ECC and ECC/ECC schemes we will compare against. The tag bits and status bits for each block are protected with the same code. Other punctured codes exist, but exploring them is beyond the scope of this paper.

B. *Using Punctured Codes in the PERC*

The organization of the PERC, as illustrated in Figure 1, is similar to that of the L1D. It has the same number of frames and the same set associativity. For each data word in the L1D, the PERC has a corresponding ECC_p code. The difference is that each block in the PERC has one more field than the corre-

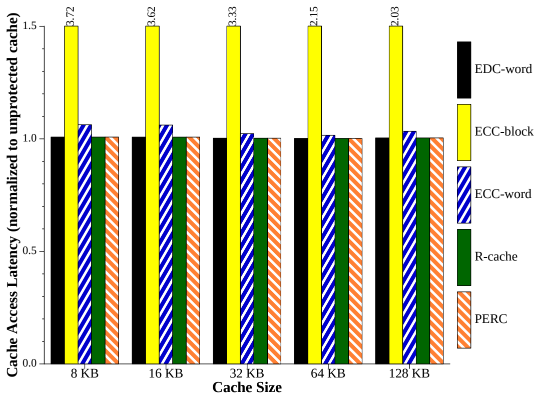


Fig. 2. L1D access latency as function of cache size

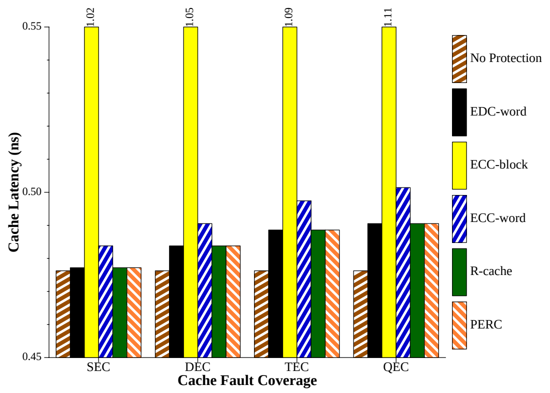


Fig. 3. L1D access latency as function of error detection/correction capability

sponding block in the L1D, since it must hold the ECC_P for that block's tags and state. In Figure 1, we show a system in which the data word and the tag/state happen to be the same size. The number of EDC_p and ECC_p bits (r_d and r_c, respectively) are a function of the desired amount of error detection/correction capability.

The operation of the PERC is as follows:

- Load hit in L1D: Read data and EDC_p from L1D. If EDC_p indicates error, then recover (see below).
- Store hit in L1D: Write data and computed EDC_p to L1D; write computed ECC_p to PERC.
- Replacement from L1D to L2: Read data word and EDC_p from L1D and read ECC_p from PERC. Write this combined block into L2. Assumes that ECC_{np} is concatenation of EDC_p and ECC_p.[1]
- Fill from L2 to L1D to satisfy L1D miss: Read block from L2, placing data word and EDC_p in L1D and placing ECC_p in PERC.
- Recover from error detected during L1D load: Read ECC_p from PERC and use it with data block and EDC_p to correct error; provide corrected data word to processor; store corrected data and EDC_p to L1D; store ECC_p to PERC.

The key to achieving our goals is that the PERC allows us to keep the ECC_p bits out of the L1D. With PERC we can use stronger ECC codes than with ECC/ECC while achieving the same performance as an ECC/ECC system with less error correction capability, since the ECC_p bits are in the PERC and thus do not impact the L1D access latency. Moreover, we can save power and die area by not keeping status bits in the PERC, since they will always mirror those in the L1D. PERC does not affect the L1D access time except by adding extra capacitive load to the data bus that carries store data to the L1D. The PERC is a write-only structure except during L1 writebacks and in the uncommon case that an error is detected by the EDC_p. Only then do we read from the PERC. In this scenario, we must wait for all outstanding writes to the PERC to complete before recovering the data. Since replacements are fairly uncommon and error recovery is rare, this small amount of waiting is unimportant.

TABLE I. Target System Parameters

clock frequency	3 GHz
pipeline depth	12 stages
pipeline width	4
instr fetch buffer	40 entries
instr window	64 entries
load-store queue	64 entries
reorder buffer	128 entries
functional units	4 integer ALUs (1 cycle), 1 integer mult/div (7/12), 2 load/store units (1)
floating point units	4 FP ALUs (4 cycles), 1 FP mult/div (4/12)
branch predictor	gshare: BHT is 4096 entries, BHT entry is 2-bit counter, BHR is 8 bits
registers	192
L1 D-cache	64K total size, 2-way, 32B blocks, 2 ports, 3-cycle latency
L1 I-cache	64K total size, 2-way, 32B blocks, 2 ports, 3-cycle latency (pipelined)
L2 cache	1M total size, 16-way, 64B blocks, 1 port, 8-cycle latency (pipelined)
main memory	150-cycle latency

IV. Experimental Evaluation

We compare the following five single-error correcting schemes to each other and to an unprotected cache (denoted *Unprotected*): EDC/ECC on the word granularity (*EDC-word*), ECC/ECC on the block granularity (*ECC-block*), ECC/ECC on the word granularity (*ECC-word*), an 8-entry *R-cache* [16], and *PERC*. We evaluate these schemes in the common error-free case, since errors are rare and thus have negligible impact.

1. Many punctured error codes exist for which this is true.

A. Methodology and System Model

We use CACTI 3.0 [10], modeling 90nm technology, to determine the latency and power consumption of the cache structures. Using the cache latencies generated by CACTI, we simulate the microprocessor using SimpleScalar 4.0 [4]. We simulate all of the SPEC CPU benchmarks with the reference inputs, and we use single SimPoints [14] (100M instructions) to sample their executions. Table I describes our microprocessor.

B. Performance

In Figure 2, we plot L1D latency as a function of the cache's size, and we normalize the latencies to those of unprotected caches. When we refer to a cache as being 64 KB, for example, we mean that the actual data blocks comprise 64 KB. The tags, state, and error code bits are not counted as part of that 64 KB, but these extra bits affect the cache's access latency, power, and area.

We observe that access latencies are similar for all schemes except ECC-block. As expected, the cache access latencies for the R-cache and PERC are similar to that of EDC-word, which is less than a 1% increase compared to Unprotected.

ECC-word is less than 10% slower than Unprotected because the access latency is generally determined by the data array's word-line decoder [2]. As we add bits to each word for ECC-word, the word-line becomes longer and its capacitance increases. This extra capacitance slows the access somewhat, but not greatly.

The one exception to the cycle time being determined by the word-line decoder is ECC-block. For ECC-block, the cycle time is determined by the circuit driving the multiplexors which select which way should be returned. Because the entire block must be read from the cache, every bit in the block needs to be multiplexed.

We previously claimed that PERC can offer stronger error protection than ECC-word with the same L1D access latency, because the ECC_p bits are in the PERC instead of the L1D. In Figure 3, we plot the absolute L1D access latency for various error correction strengths. In the figure, SEC, DEC, TEC, and QEC refer to Single, Double, Triple, and Quadruple Error Correction, respectively. For example, QEC can correct all errors of 4 bits or less. The results show that PERC with DEC has similar access latency as ECC-word with SEC, and PERC with QEC has similar latency as ECC-word with DEC. As transient error rates continue to increase and stronger error correction codes become necessary, this advantage of PERC could become important.

The impact of L1D latency on overall microprocessor performance is shown in Figure 4. Differences between benchmarks are mostly due to differing demands on the L1D and differing abilities to overlap useful work while waiting for cached data. As expected from the results in Figure 2, ECC-word and PERC have comparable performance to Unprotected. Also unsurprisingly, ECC-block performs poorly. The other schemes, even those with comparable cache access latencies, sometimes suffer in overall performance because of the extra bandwidth demands they place on the L2 cache. EDC-word and the R-cache, on average, only degrade performance by about 5-10%, but the slowdown is as much as 50% on the benchmark *apsi*.

C. L2 Cache Bandwidth

One motivation for PERC is that it reduces pressure on the L2 cache by not requiring a write-through L1D like EDC-word. In Figure 5, we plot the mean L2 bandwidth for each benchmark for error protection scheme. Neither ECC scheme nor PERC requires any additional L2 bandwidth. In fact, ECC-block uses far *less* bandwidth, because its performance is so much worse (i.e., it transfers the same number of bytes to the L2 but over a much longer time period). EDC-word and the R-cache require substantially more L2 bandwidth than Unprotected. These increases in L2 bandwidth represent either a large cost (more L2 ports) or a decrease in performance due to contention for the L2. In general, the benchmarks with the greatest increases in L2 bandwidth exhibit the most performance degradation, but some benchmarks with very low Unprotected L2 usage (e.g., *crafty*) can tolerate a large percentage increase.

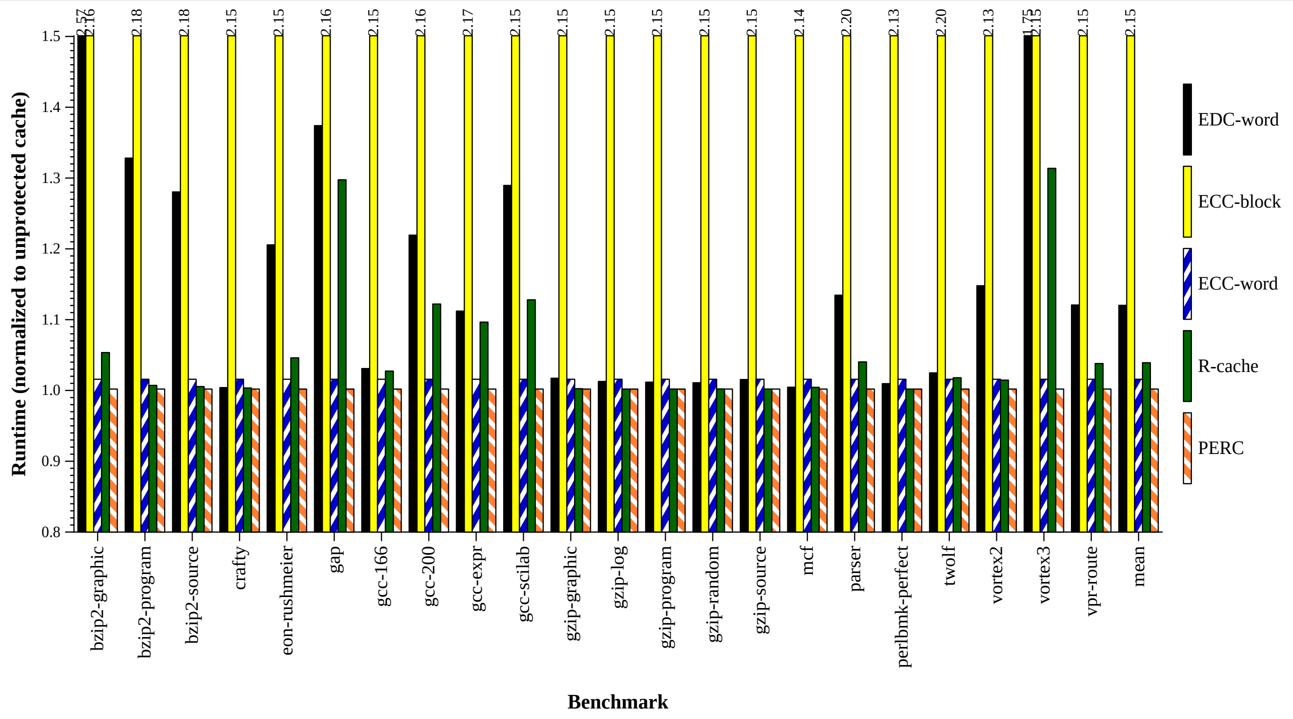
D. Power Consumption

In Figure 6, we plot the common-case, error-free power usage per L1D access of each scheme as a function of the cache size, normalized to the power of Unprotected. For EDC-word, this power includes the average additional power used for write-throughs to the L2. To compute this, we use the mean fraction of stores, which is 24.9% of all cache accesses, and multiply this by the power consumed by an L2 access. For the R-cache, the power includes the power used by the R-cache itself, as well as power used for additional writebacks to the L2 (beyond those performed by Unprotected). For PERC, the power includes the power used by the PERC for stores, replacements from the L1D to the L2, and fills from the L2 to the L1D.

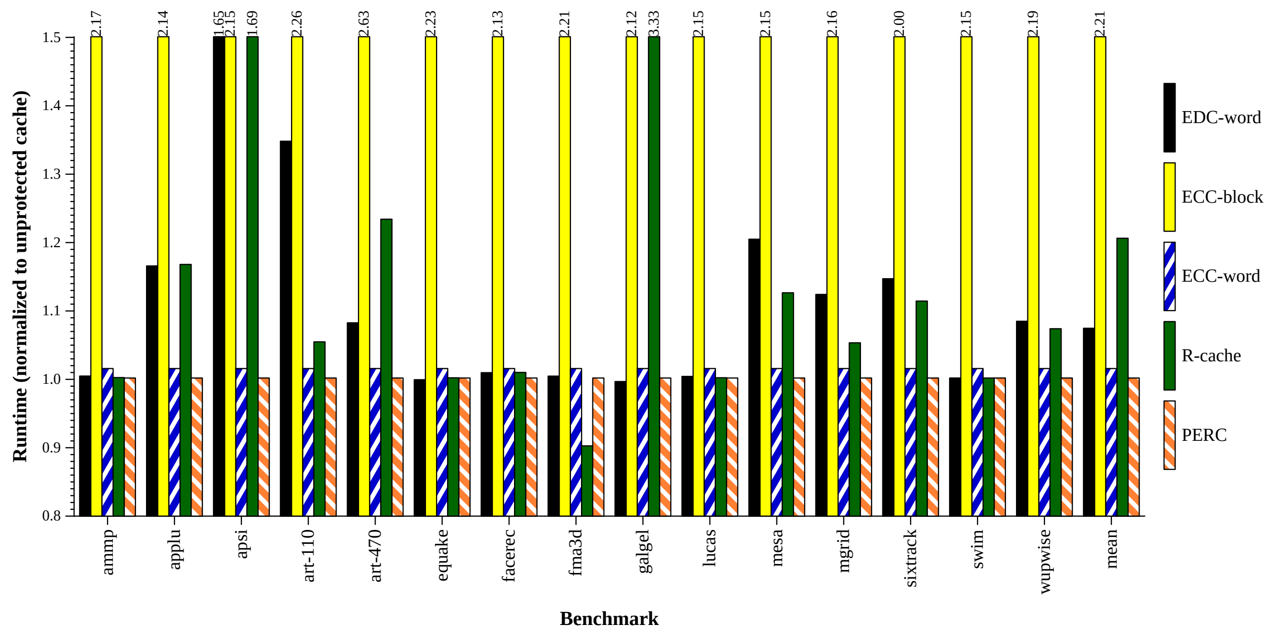
ECC-word and PERC use power that is comparable to Unprotected, although PERC consistently uses less power than ECC-word. EDC-word uses more power because of the additional write-throughs to the L2. ECC-block uses more power than these two schemes because of how many bits need to be accessed. The R-cache uses 30-61% more power than Unprotected just for R-cache accesses, even though it is a small structure, due to its full associativity. This corroborates research which has shown that fully associative structures can use as much as five times as much power as similarly sized RAMs [8]. Moreover, the R-cache scheme uses a significant amount of extra power at the L2.

E. Die Area

While area is generally not the most critical resource in most systems, we want to make sure that no design uses an egregious amount of area. In Figure 7, we plot the area (normalized to Unprotected) required to store the L1D data and error protection bits for a cache that can tolerate single-bit errors, as a function of the cache size (including auxiliary structures, such as the R-cache or PERC). We see that ECC-block is usually the largest (except for the 64KB cache size) due to the need to output more bits. Conversely, EDC-word has



(a) SpecINT benchmarks



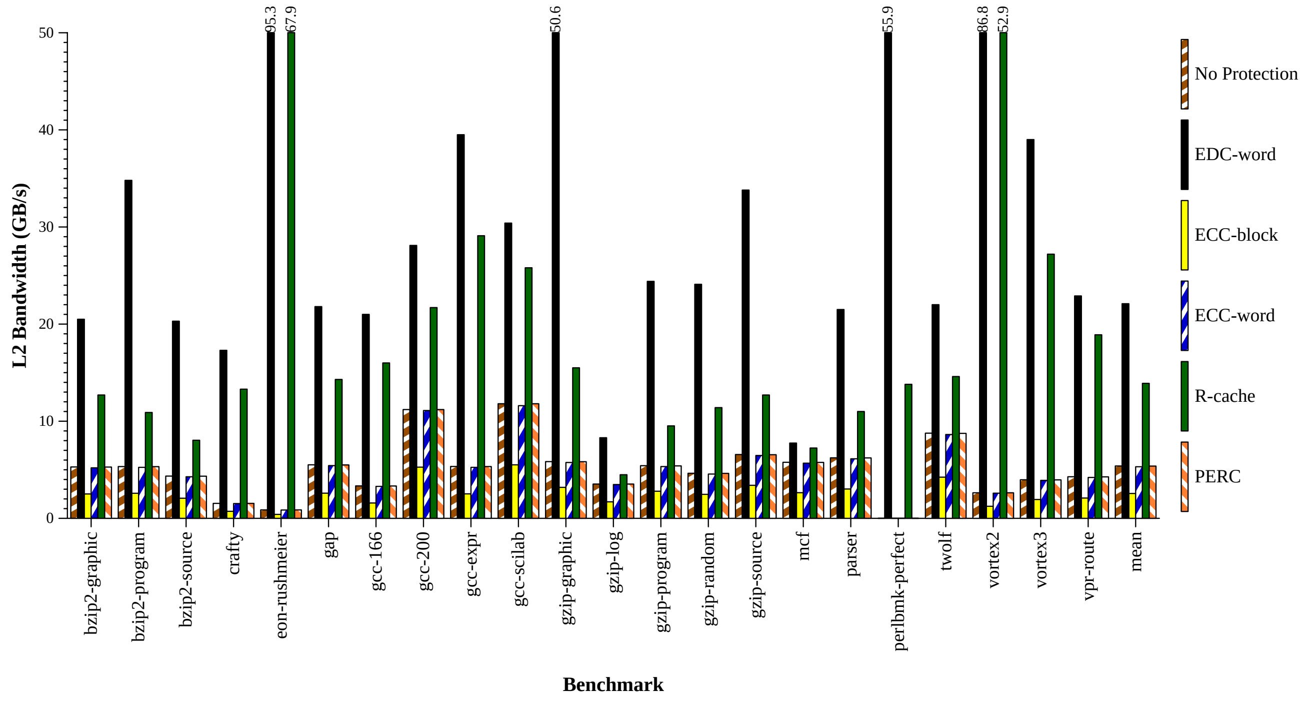
(b) SpecFP benchmarks

Fig. 4. Impact of Error Protection on Overall Runtime

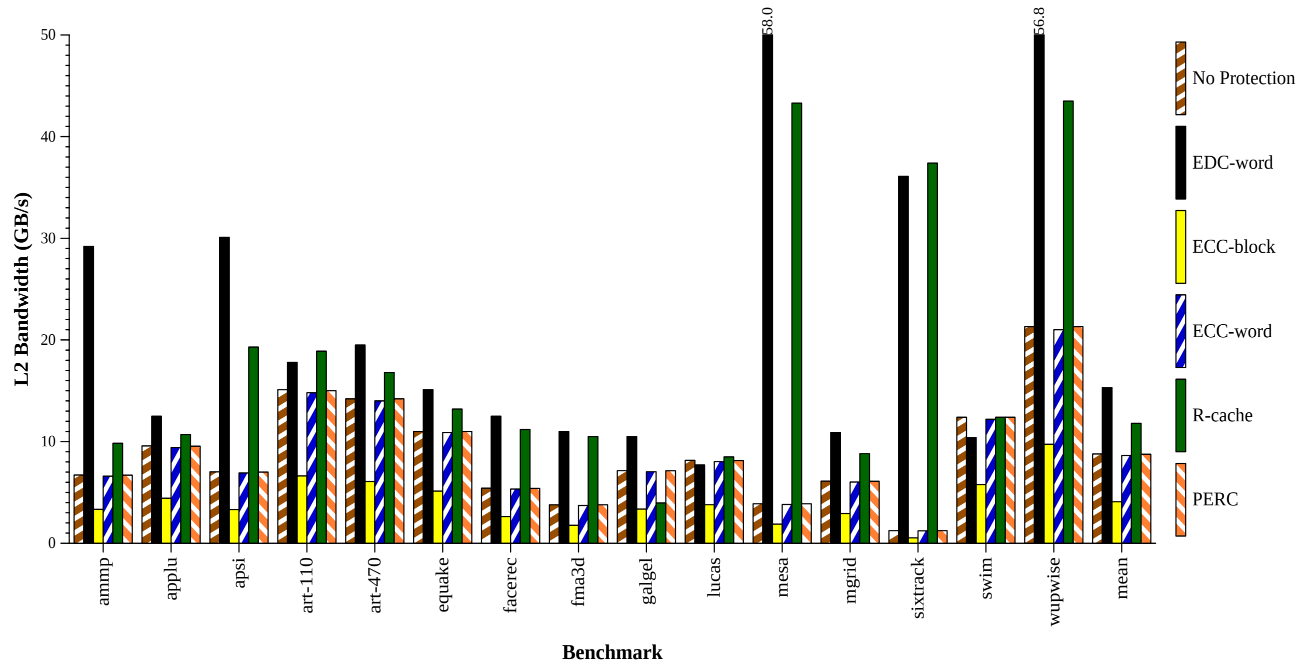
a very small overhead of just one bit per word. In between these extremes, the relative overheads of the other three schemes vary based on the size of the cache. We believe these results show that any of these schemes are acceptable from an area consideration.

V. Conclusions

Our experimental results show that using ECC on the word granularity in the L1D cache is almost always preferable to the other prominent technique of using EDC on the L1 with ECC on the L2. Our evaluation of PERC shows that it achieves per-



(a) SpecINT benchmarks



(b) SpecFP benchmarks

Fig. 5. L2 Cache Bandwidth

formance that is comparable to these two existing approaches, while using the same bandwidth as ECC (which is the same as for an unprotected cache) and slightly less power than ECC. Moreover, PERC can offer performance advantages over ECC/ECC in three scenarios: (1) if stronger error correction codes are desired, (2) if the additional bits for ECC (as compared to EDC) would require a change in the geometry of the L1D, because this change in geometry could significantly slow down the L1D, and (3) if, for future technologies, the additional bits for ECC (as compared to EDC) cause significant slowdown.

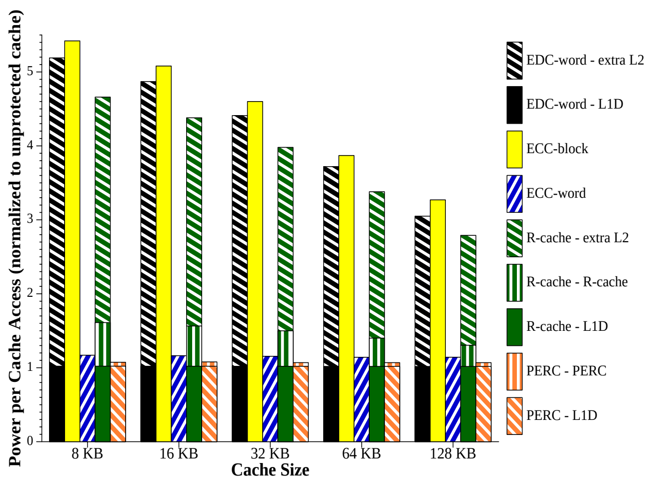


Fig. 6. Power Consumption per Cache Access (normalized to unprotected cache)

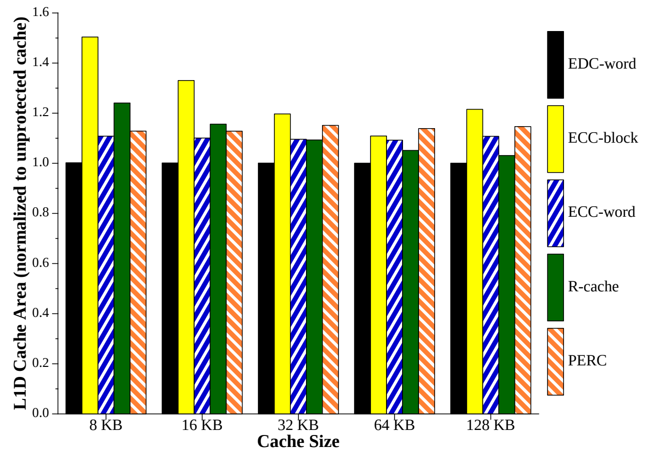


Fig. 7. Chip area required for L1 data and error protection (normalized to unprotected)

Acknowledgment

This work is supported in part by the National Science Foundation under grants CCF-0444516 and CCR-0309164, the National Aeronautics and Space Administration under grant NNG04GQ06G, Intel Corporation, and a Warren Faculty Scholarship. We thank Alvin Lebeck and Jaidev Patwardhan for their helpful comments on this work.

References

[1] Advanced Micro Devices. AMD Eighth-Generation Processor Architecture. Advanced Micro Devices Whitepaper, Oct 2001.

[2] B. S. Amrutur and M. A. Horowitz. Fast Low-Power Decoders for RAMs. *IEEE Journal of Solid-State Circuits*, 36(10):1506–1515, Oct 2001.

[3] G.-H. Asadi et al.. Balancing Performance and Reliability in the Memory Hierarchy. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 269–279, Mar. 2005.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[5] D. C. Bossen, J. M. Tendler, and K. Reick. Power4 System Design for High Reliability. *IEEE Micro*, 22(2):16–24, Mar/Apr 2002.

[6] G. I. Davida and S. M. Reddy. Forward-error Correction with Decision Feedback. *Information and Control*, 21(2):148–170, Sept. 1972.

[7] J. Du, M. Kasahara, and T. Namekawa. Separable Codes on Type-II Hybrid ARQ Systems. *IEEE Transactions on Communications*, 36(10):1089–1097, Oct. 1988.

[8] A. Efthymiou and J. D. Garside. An Adaptive Serial-Parallel CAM Architecture for Low-Power Cache Blocks. In *Proc. of the 2002 Int'l Symposium on Low Power Electronics and Design*, pages 136–141, 2002.

[9] Intel. *Intel Pentium 4 Processor on 90 nm Process Datasheet*. Intel Corporation, Apr 2004.

[10] N. P. Jouppi and S. J. Wilton. An Enhanced Access and Cycle Time Model for On-Chip Caches. DEC WRL Research Report 93/5, July 1994.

[11] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[12] S. Kim and A. K. Somani. Area Efficient Architectures for Information Integrity in Cache Memories. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 246–255, May 1999.

[13] L. Li et al. Soft Error and Energy Consumption Interactions: A Data Cache Perspective. In *ISLPED '04: Proc. of the 2004 International Symposium on Low Power Electronics and Design*, pages 132–137, 2004.

[14] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[15] Sun Microsystems. UltraSPARC IV Processor Architecture Overview. Sun Microsystems Technical Whitepaper, Feb 2004.

[16] W. Zhang. Enhancing Data Cache Reliability by the Addition of a Small Fully-Associative Replication Cache. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 12–19, June 2004.

[17] W. Zhang et al. ICR: In-Cache Replication for Enhancing Data Cache Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 291–300, June 2003.