LAEC: Look-Ahead Error Correction Codes in Embedded Processors L1 Data Cache

Pedro Benedicte*†, Carles Hernandez*, Jaume Abella* and Francisco J. Cazorla*‡
* Barcelona Supercomputing Center † Universitat Politècnica de Catalunya ‡ IIIA-CSIC

***Abstract*—As implementation technology shrinks, the presence of errors in cache memories is becoming an increasing issue in all computing domains. Critical systems, e.g. space and automotive, are specially exposed and susceptible to reliability issues. Furthermore, hardware designs in these systems are migrating to multi-level cache multicore systems, in which write-through first level data (DL1) caches have been shown to heavily harm average and guaranteed performance. While write-back (DL1) caches solve this problem they come with their own challenges: they need Error Correction Codes (ECC) to tolerate soft errors, but implementing DL1 ECC in simple embedded micro-controllers requires either complex hardware to squash instructions consuming erroneous data, or delayed delivery of data to correct potential errors, which impacts performance even if such process is pipelined. In this paper we present a low-complexity hardware mechanism to anticipate data fetch and error correction in DL1 so that both (1) correct data is always delivered, but (2) avoiding additional delays in most of the cases. This achieves both high guaranteed performance and an effective solutions against errors.**

I. Introduction

Critical embedded systems have seen a rapid increase in their (guaranteed) performance requirements in recent years. This trend is fueled by the use of complex artificial-intelligence based software to handle huge amounts of data, e.g. coming from cameras, and implement autonomous driving functionality [8]. At hardware level, this has caused a fast transition from simple 8- and 16-bit single-core micro-controllers to SoC designs encompassing multicore processors equipped with several cache levels. A common processor design in automotive [1], avionics and space [5] is to have private first level caches that access a shared L2 cache via a communication means (e.g. a bus). Previous studies show that contention in the access to hardware share resources can increase tasks worst-case execution time (WCET) significantly [14], [16]. In this line, the use of write-through DL1 caches has a huge negative impact on performance since every store instruction accesses the shared communication hardware. The effects on WCET intensify since time allowances to capture worst case scenarios. In particular, write-through DL1 can increase WCET up to 6x just for the bus contention compared to write-back designs [9].

The sensitivity of caches to errors (faults) is another issue of up most importance critical systems. Critical systems must undergo a strict certification process to provide evidence that specific failure rates are below specific thresholds set in applicable safety standards, e.g. ISO26262 [21] in cars. Critical systems must include safety mechanisms for fault tolerance to ensure low-enough acceptable failure rates.

From the previous discussion it follows that chip designers for critical systems face conundrum in the design of DL1 caches to provide both reduced WCET estimates (high guaranteed performance) and keep low rates under control. On the one hand, instruction (read-only) caches and write-through DL1 never keep a dirty copy of any data. Hence, they can implement low-cost error detection mechanisms such as parity, since error-free copies of the data exist elsewhere (e.g. in the L2 that is ECC protected). This however comes at the cost of increased WCET estimates. On the other hand, write-back or hybrid write-through/write-back DL1 caches [9] contain the impact of contention in WCET estimates by avoiding that every store access shared resources. DL1 write-back caches design may keep dirty data and hence, error correction means are needed to keep failure rates low enough. However, tolerating faults in DL1 cache memories requires, in general, the use of Error Correction Codes (ECC) to allow recovering data that has been corrupted, which carries significant impact in either DL1 cache latency or design complexity. If DL1 cache data is delivered before correction, ECC does not impact the critical path and can be computed offline. However, upon the detection of an error, direct and indirect consumers of erroneous data are squashed and restore a correct state before resuming operation. In general, simple microcontrollers used for critical real-time systems lack such support. If cache data is delivered after correction, an additional stage is needed after loading data to compute ECC and validate whether data is correct. Thus, back-to-back execution of consumers after a load operation occurs with an additional delay (typically one cycle), which has non-negligible impact in performance.

In this paper, we present an alternative deployment of ECC in L1 caches for critical real-time microcontrollers aimed at mitigating the impact of ECC calculation in L1 caches. We propose a Look-Ahead Error Correction (LAEC) scheme that anticipates the whole DL1 access process by one cycle, thus allowing to eliminate any performance overhead whenever such anticipation is possible. In particular, in in-order cores common for critical real-time embedded systems (e.g. LEON4, ARM Cortex-R5 cores), whenever the input address registers are not computed by the immediate predecessor instruction of the load instruction and the predecessor instruction is not a load instruction, we can perform address calculation, DL1 access and ECC computation one cycle ahead of time. In this way, data can be delivered in the same cycle it would be delivered in a non-protected DL1 cache without such anticipation. We note that the constraints that could preclude the effectiveness of our mechanism occur seldom, thus allowing our approach achieving a performance very close to that of an error-free processor without ECC, and outperforming designs that require an additional cycle before delivering error-free data.

We have evaluated our proposal implementing the proposed look-ahead error correction scheme with Single-Error Correction Double-Error Detection (SECDED) in the DL1 cache of a cycle accurate processor model of the LEON4 [5]. Our results show that our look-ahead error correction scheme outperforms

TABLE I: Commercial processors and their characteristics.

Processor	Frequency	L1 WT	L1 WB
ARM Cortex R5	160MHz	Yes, ECC/parity	Yes, ECC/parity
ARM Cortex M7	200MHz	Yes, ECC	Yes, ECC
Freescale PowerQUICC	250MHz	Yes, Parity	Yes, parity
Cobham LEON 3	100MHz	Yes, parity	No
Cobham LEON 4	150MHz	Yes, parity	No

the baseline read-and-correct scheme by 6% on average across EEMBC Automotive [28] benchmarks, and is within 3.9% the performance of an ideal error-free design without any ECC support.

II. Motivation

Current processor designs for critical systems employ different approaches to include ECC schemes in caches. This is partially motivated by the fact that actual latency overheads depend on the particular ECC technique employed. For instance, using a parity bit is the simplest and fastest technique, and SECDED a more complex and slower one. In general, processors targeting safety critical systems require having the ability to recover from faults and this forces processor designers to architect solutions able to achieve that goal. As shown in Table I processors available in the market use different approaches to protect caches from errors. For instance, the LEON family of processors advocates for using write-through caches with parity in the L1. The Freescale PowerQUICC offers the user the possibility to configure L1 caches as write-through or write-back restricting recovery capabilities to write-trough configuration only. They pay the costs in contention to reduce faults, since in the space domain these are more common. Finally, in the Arm Cortex family the processor IP is sold with the possibility of implementing both write policies and allowing using ECC or parity in L1 caches. However, as acknowledged in the datasheet [6], using ECC in the L1 can impact the maximum operating frequency of the processor. In this case the final decision on whether to tradeoff performance for reliability is left to the integrator.

A. Correcting Errors in Write-Through (WT) DL1 caches

A practical solution typically found in processors for critical system consists of using a cache hierarchy with inclusive caches and write-through (WT) policy in the DL1 [5], [7], [29]. A commonality in these designs is to include a parity bit in DL1 caches and SECDED in the L2 cache since the impact of latency overhead of SECDED in the L2 is lower. The main reason is that, even if L2 read hit latencies are increased due to the introduction of SECDED, its impact in overall performance is low due a two-fold reason. First, L2 read accesses occur seldom, and second, having an additional L2 cycle causes limited impact due to the already high L2 access latencies to send requests, access the L2 itself and return data read to the core. Overall, this configuration (DL1 parity + L2 SECDED) ensures that errors can be detected with the parity bit with virtually no impact in latency in the DL1, and recovered with the SECDED mechanism implemented in the L2.

While configurations using WT caches offer a workaround to the problem of correcting data in the DL1, this configuration presents the drawbacks that are inherent to the use of WT caches such as lower performance and higher energy consumption since every store operation is always propagated from the DL1 to the upper levels of the memory hierarchy (i.e. hardware shared resources). To mitigate this issue processors may include a store-buffer and/or use an L2 cache implementing a write-back (WB) policy. However, it has been shown [9] that performance guarantees (WCET estimates) on multicore processors incorporating WT caches are quite poor when compared with their WB counterpart despite implementing store-buffers and a WB L2 cache. This result is especially important since processors targeting critical systems do not only require guaranteeing reliable operation, but also offering high performance and time-predictable behavior [21], which calls for multicore processors implementing ECC in WB DL1 caches in an efficient manner.

B. Correcting Errors in Write-Back (WB) L1 caches

WB policies do not update, on a DL1 hit, the upper levels of the memory hierarchy. Hence, in our setup modified data can reside exclusively in the DL1, so using a WB policy requires implementing error correction capabilities to recover from errors in the DL1. However, as explained before, this has generally an impact in the access time to DL1 cache. There exist several approaches to deal with the increase in DL1 access latency already implemented in commercial processors. The particular processor architecture, the target operating frequency, and the manufacturing technology determine when to use one approach or the other. Below we describe four existing approaches:

1) **Decrease the operating processor frequency** is the most trivial approach to allow SECDED in the DL1 so that the error correction process can be accommodated within the last cycle of the cache access. However, this has a significant impact in the performance of the system. Some commercial processors for which the targeted operating frequency is sufficiently low or whose critical path is determined by other components may opt for this solution [6].
2) **Extra cache cycles.** Adding extra (non-pipelined) cycles in the DL1 access so that ECC computation fits in the L1 cache access time without impacting operating frequency. However, such a solution virtually doubles the time utilization of the DL1.
3) **Extra stage.** Pipelining cache accesses such that instructions proceed normally, adding a final cycle for ECC computation. Pipeline stalls will be introduced in the case of data dependencies (i.e. an instruction requires data for which ECC computation is not yet performed). The delay of the logic that detects and corrects errors can vary depending on the number of bits corrected. For SECDED, considered in this paper, this latency is smaller than an DL1 cache access [13], [18], and thus fits in a single additional cache cycle or stage pipeline.
4) **Speculate and flush.** Using a speculate and flush approach consists of processing accesses and delivering unchecked data, which may be used in parallel with ECC computation. Whenever the result of the ECC determines that the propagated data was erroneous, the pipeline is flushed or some instructions squashed, and a previous correct state needs to be recovered.

From these four approaches, we discard the former due to its noticeable performance degradation, and the latter due to the implementation complexity required to implement a flush mechanism in simple microcontrollers for critical systems

like the ones we target. The extra stage and cache cycles solutions offer acceptable cost and implementation complexity trafeoffs, and will be the reference policies we compare our proposal with. Our proposal builds upon the Extra stage one, but anticipates the load access and ECC computation whenever possible so that no additional stalls are introduced due to ECC computation. In the next section we introduce details of the baseline approaches and present our look-ahead scheme.

III. Look-ahead Error-Correction

We propose an alternative approach to deploying ECC in DL1 caches. It consists in anticipating one cycle the address computation, the load access, and the ECC computation. This can be done when no data or structural dependence with older instructions occurs. Effectively anticipating one cycle the processing of DL1 load hits allows anticipating ECC computation by one cycle too. As a result, an instruction dependent on the loaded data can be executed back-to-back with the load without experiencing any delay due to ECC computation.

In this section, we first introduce the mechanism used to anticipate the address in our Look-Ahead Error-Correction (LAEC) proposal. Then, we describe the processor model where the implementation and experiments will be conducted. Finally, we describe the implementation details of Extra Cache Cycle and Extra Stage approaches, as well as LAEC.

A. Address anticipation mechanism

There are several ways to predict the address of the next access to cache. For instance, cache designs could incorporate a predictor similar to the ones employed in hardware data prefetchers [15]. However, since the focus of this paper is deploying ECC in relatively simple processors, we opt for an alternative method to predict the next DL1 access address. LAEC avoids mispredictions by anticipating address calculation only when it is guaranteed that such anticipation will deliver correct results.

In particular, LAEC avoids speculating on the address to prevent unnecessary accesses to the DL1. We anticipate address computation by reading the base register one cycle earlier if it has not to be modified by any previous instruction. This allows the address of the access to be computed one cycle earlier using an adder to add the register and the offset. LAEC also requires including two extra ports to the register file to retrieve the registers one cycle earlier, but in general an in-order single-issue processor has a small register file with few ports, so this would incur low power cost. In fact, it has been shown that energy is largely dominated by cache memories, so the energy consumption of the register file is tiny [26]. An access look-ahead can be performed when following two conditions hold:

1) *No resource hazard*. Since we anticipate the DL1 access and ECC computation by one cycle, we may conflict with the previous instruction if it accesses the DL1 simultaneously with the anticipated instruction. This occurs when the previous instruction is a non-predicted (i.e. branch speculated) load.
2) *No data hazard*. When the instruction prior to the load produces the address register of the load, we cannot anticipate the address computation. This is so because the input data for the ongoing instruction (load) is not yet ready whenever we want to anticipate its execution.

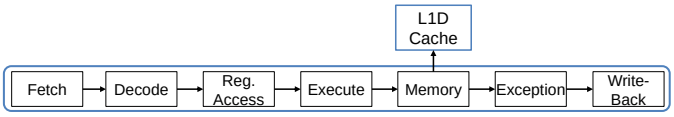


Fig. 1: Baseline NGMP-like processor pipeline.

If none of these hazards occur, then we can compute the address, access DL1, and compute the ECC one cycle ahead of time. With *no resource hazard*, we guarantee that the DL1 read port is available. With *no data hazard* we guarantee that we are loading the right data, so no misprediction can occur and there is no need to flush.

B. Processor Model

In order to implement LAEC in the DL1, we use a system resembling the NGMP [5]: a multicore processor that includes 4 single-issue in-order pipelined cores with L1 private caches and a shared L2 cache. In the NGMP, error recovery is guaranteed by using WT DL1 caches with a parity bit and implementing SECDED in the WB shared L2. However, to implement LAEC we modify the baseline implementation to include a WB DL1 cache. Note that this modification is already in the roadmap of the LEON processor family whose providers have already announced LEON5 processor implementing WB DL1 caches [11]. Also, using a WB DL1 cache has already been proven effective for this setup [9].

The original NGMP system has a seven stage pipelined design (see Figure 1). The memory stage uses a write buffer (not shown in Figure 1 for simplicity) where all writes are stored until they can access DL1. A load that misses in DL1 blocks the pipeline. All loads stall the memory stage until the write buffer is empty to avoid consistency issues. Writes also stall the pipeline with backpressure when the write buffer is full, until it gets completely empty.

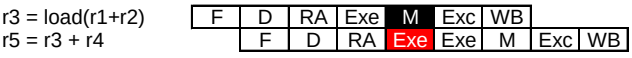


Fig. 2: Chronogram of a data dependency stall on the NGMP.

In Figure 2 we show an example of two consecutive instructions with a data hazard between them that results in a 1 cycle stall for the younger instruction. In red we show the stage in which the young instruction stalls and in black the stage where the DL1 cache is accessed. In this case, it matches the memory stage, but this is not always the case in our proposed approach.

Next, we present the implementation details for existing Extra Cache Cycle and Extra Stage solutions as well as for LAEC. We also show how they could be implemented in a processor like the NGMP.

C. Extra Cache Cycle Implementation

A first simple approach that would require little changes to the current architecture is to make the ECC check in the memory stage so that this stage spans across two cycles, thus increasing the latency of a load hit from 1 cycle to 2 cycles.

In terms of hardware cost and implementation complexity, besides the ECC logic and its associated array in the DL1, little extra logic is needed in order to stall earlier processor stages

(those before the memory stage), since stall logic already exists to stall the pipeline upon a DL1 cache miss.

The performance impact that this solution can have is relatively high, since it will double the cycles in the memory stage for DL1 hits.

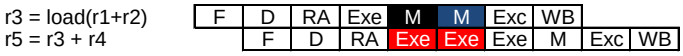


Fig. 3: Data dependency stall with Extra Cache Cycle.

Figure 3 extends the example in Figure 2 when the Extra Cache Cycle solution is applied. Now the young instruction that depends on the loaded data needs to stall one additional cycle for the value to be both loaded and checked. The stage in blue performs the ECC computation, which is done on the second cycle of the memory stage.

D. Extra Stage Implementation

Another simple approach would be to add a new pipeline stage after the Memory one: the ECC stage. This stage would compute the ECC for DL1 load hits, and compare it with the existing value stored in its ECC array. For writes that hit, it would compute the new ECC and store it in the ECC array.

In terms of timing, this solution can stall the pipeline when a load that hits in cache is followed (distance 1 or 2) by an instruction that uses the loaded value. In particular, the instruction immediately after the load cannot use the loaded value because its execute stage overlaps with the load memory stage. The second instruction starts its execution stage right after the load fetches the data from DL1 on a cache hit. Hence, if the second instruction after the load was allowed to use the loaded value as a source operand before computing its ECC and this value was incorrect, a complex recovery mechanism would be required to restore the processor state to a previous correct state. Instead, in our implementation, if this scenario happens, the processor stalls to avoid continuing the execution with a potential incorrect value.

It is worth noting that this only happens for loads that hit on the DL1 cache, because for loads that miss and have to request the data to higher levels (L2 or memory) the ECC of the data is checked in the corresponding cache level or main memory, so there is no need to check it again in the new ECC stage. For stores, the write buffer is usually enough to hide this latency.

In addition to the ECC logic and ECC array in DL1, small extra hardware is needed in order to stall the stages before the Memory one. However, as explained before, this logic already exists to manage DL1 load misses.

In terms of potential impact, this solution is affected negatively by the number of times that an instruction consuming the loaded data is stalled due to the ECC stage, which can occur often since it is common having a consumer for loaded data in the range of the 2 following instructions.

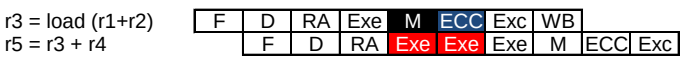


Fig. 4: Data dependency stall with Extra Stage.

Figure 4 shows a scenario similar to Extra Cycle. The young instruction needs to stall for 2 cycles due to the data hazard. The advantage over the previous solution is shown in Figure 5: when there is no data hazard, consecutive instructions can continue execution without a stall, since Memory and ECC are pipelined.

Fig. 5: No data dependencies with Extra Stage.

E. LAEC implementation

LAEC anticipates the load and ECC computation by 1 cycle. To that end, the address registers are read one cycle earlier, so two additional read ports are required in the register file. If any of the registers has been generated but not yet stored in the register file, it can be obtained from existing bypasses. Since the address computation for the load needs to be performed one cycle earlier (RA stage), an additional adder is also required (see Figure 6). We have checked with CACTI [17] the access times of a register file and an DL1 cache like the ones found in the LEON4 [5] (1088 bits for the register file, 16KB for the DL1 in 65nm). The difference between both is enough to include a 32 bit adder [2], so this addition would not increase the stage time of the memory stage.

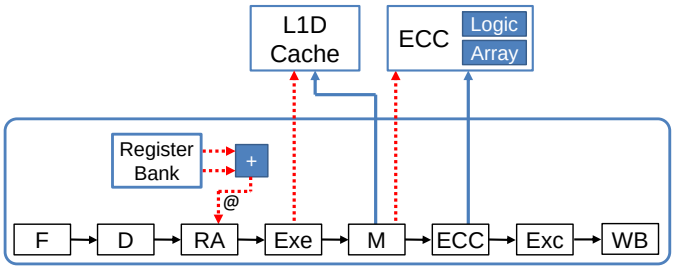


Fig. 6: Modified NGMP-like processor to support ECC using LAEC.

If the previous instruction generates one of the source operands of the load instruction, this technique cannot be used, since it would require the operands a cycle earlier than they are available. In this case, the processor operates normally (like in the Extra stage implementation), with no look-ahead. Then, if any of the 2 instructions right after the load requires the loaded data, there will be a cycle penalty due to the address not being previously computed. Analogously, if the previous instruction is a non-predicted load, it will require the DL1 port (memory stage) simultaneously with the current anticipated load. In this case, the current load cannot be anticipated due to a resource hazard. These two scenarios are the only ones where our solution can have a penalty. This means that our look-ahead proposal will always perform equal or better than the Extra stage implementation since, in the worse case, it cannot anticipate the load and just operates the same way as the Extra stage.

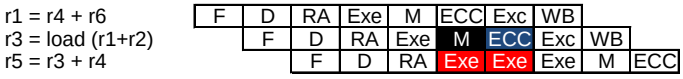
Figure 7 (a) shows a scenario where the added ECC penalty cycle can be avoided because of prediction. Registers r1 and r2 are both read and added on the RA stage. Then, on the Exe stage, the DL1 cache is accessed. Afterwards, on the M stage the ECC is computed. This results in the loaded data being ready to the younger instruction without additional penalty when compared to the baseline no-ECC solution (Figure 2). Conversely, Figure 7 (b) shows a scenario where there is still penalty due to a data dependency that prevents the look-ahead. Now the load is preceded by an instruction that computes one of its source registers. This means that r1 is not ready in the RA stage, so a normal execution (DL1 access on M stage and

TABLE II: Performance impact of existing approaches.

	a2time	aifftr	aifirf	aiifft	basefp	bitmnp	cacheb	canrdr	idctrn	iirflt	matrix	pntrch	puwmod	rspeed	tblook	ttsprk	average
% of hit loads	89	97	90	97	84	98	77	86	92	86	99	90	85	84	88	84	89
% of dep. loads	68	53	66	54	80	65	13	67	59	63	64	61	66	66	68	61	60
% of loads	23	21	26	21	24	20	18	29	21	26	20	25	31	29	29	31	25

ECC computation on ECC stage) is performed, resulting in a 1 cycle extra stall.

(a) Look-ahead on LAEC.



(b) Normal (no look-ahead) execution on LAEC.

Fig. 7: Possible scenarios with LAEC.

The schematic of the modifications required is shown in Figure 6. Note that the DL1 cache can now be accessed in two different stages: Exe or M. It will be accessed in Exe when there is a look-ahead (in red) and in M when there is not (in blue). Likewise, the ECC logic and array can also be accessed in two different stages: M (for look-ahead, in red) and ECC (normal execution, in blue). Since the baseline processor already includes most of the control logic needed for this solution, as well as bypasses from the desired stages, there is no significant cost in terms of hardware. The only explicit changes apart from the logic are a 32-bit adder and two extra read ports on the register file. Overall, implementing our LAEC proposal in an NGMP-like processor incurs in low hardware cost and implementation complexity.

IV. Evaluation

We use the SoCLib [30] framework that models the NGMP architecture [5]. In the base design, each core has a 7-stage pipeline (up to 8 stages with our modifications), a private L1 instruction cache and a 4-way 32B/line DL1 cache. It is a 4-core system connected with a bus to a shared L2 cache, which then connects to off-chip memory. We evaluate the EEMBC automotive [28] benchmarks, a suite of common critical real-time programs used in the automotive domain. The simulations are run in a multicore system but only a single core executes a task, since the focus of this work is on core performance.

The overhead of the existing approaches is due to stalls that happen when there is a DL1 hit that has a data dependency with the preceding instruction.

In Table II, the first row shows the percentage of load instructions that hit in the DL1 cache. We see that with an average of 89% hit rate, most of the loads generate hits in cache, hence potentially generating stalls. The second row shows the percentage of load instructions followed, at a distance 1 or 2, by an instruction that uses as a source operand the loaded data. On average 60% of the loads cause a stall. Finally, loads represent between a 20% and 30% of all the executed instructions, significant enough to impact performance.

A. Experimental results

In terms of performance, Figure 8 shows the increase in execution time with respect to a no-ECC system. Extra

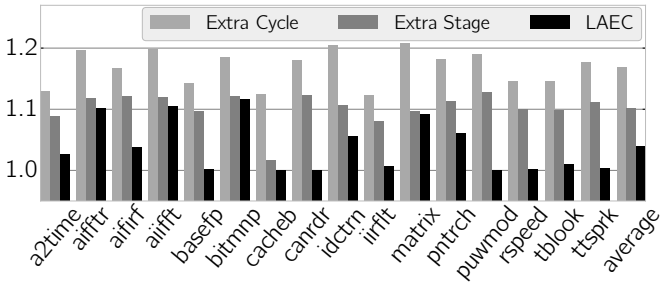


Fig. 8: Execution time increase of the different solutions compared to the baseline no-ECC system.

cycle shows the highest performance degradation, with a 17% execution time increase on average w.r.t. a configuration without ECC stage, reaching up to 20% for some benchmarks (`aifftr` and `matrix`).

Extra stage shows around 7% less performance degradation than Extra cycle, with a 10% on average. This occurs because its pipelined designed avoids some stalls. All benchmarks perform similarly, except `cacheb`. This benchmark shows little performance degradation compared with the baseline no-ECC (2%). This is due to the number of loads that are followed by dependent instructions. While in rest of benchmarks between 50% and 80% of the loads have this property, in `cacheb` just 13% of the loads have it. This results in fewer cases that stall the pipeline an additional cycle, and thus in lower performance degradation.

Finally, LAEC, due to its anticipated load execution, saves most of the stalls. On average, LAEC only increases execution time less than 4%, being such increase below 1% in several benchmarks such as `basefp`, `cacheb`, `canrdr`, `puwmod`, `rspeed` and `ttsprk`. Out of the two potential conditions that LAEC needs to meet to cause a stall (resource and data hazards), most of them are due to data hazards. That is, the scenario where an instruction generates the address to be loaded, the next instruction performs the load that hits in cache, and the next 1 or 2 instructions consume the loaded data (as shown in Figure 7 a)). There are four benchmarks (`aifftr`, `aiifft`, `bitmnp`, `matrix`) that show almost no improvement when comparing LAEC with Extra Stage. This is caused because most of the loads that have dependent instructions executed right after, so causing stalls for Extra Stage, also have their source operand produced by the previous instruction, which prevents load anticipation and causes stalls for LAEC.

In terms of power, the proposed solution has minimal impact (less than 1%). However, since the execution time is increased, leakage energy consumption increases proportionally to the increase in execution time. This means that for extra cycle and extra stage, leakage energy consumption increases by around 17% and 10% on average; and for LAEC by less than 4%.

Note that, as explained before, while compiler optimizations could help mitigating stalls, they are normally forbidden in critical software due to traceability between source and binary files needed for certification. Moreover, those systems often execute legacy code where no binary modifications are possible. On average, LAEC shows a 13% decrease in performance degradation when compared to Extra cycle and a 6% decrease compared to Extra stage.

V. Related Work

The most common microarchitectural solution to error correction relies on the use of parity or error ECC [10] to detect and correct errors. Parity suffices for read-only caches (e.g. instruction caches) and write-through caches. When data can be dirty, then ECC is required.

Several works aims at providing support for both, permanent and transient faults. Those works often consider high permanent fault rates due to low voltage operation, and propose mechanisms to tolerate those faults while providing resilience against transient faults. Some works propose disabling faulty entries at different granularities [3], [27], potentially setting up spare cache lines to replace faulty ones [23], or combining faulty entries to form fault-free ones [22], [32], potentially combining these designs with heterogeneous ECC depending on the faultiness of cache lines [33].

Some authors combine fault tolerance in caches with real-time requirements by ensuring that ECC guarantees that non-correctable permanent fault rates are below specific thresholds [24], [25], or by guaranteeing that spares (in the form of a victim cache) suffice to guarantee sufficiently low non-correctable permanent fault rates [4].

Some techniques target specifically soft errors. Some authors propose early evicting dirty cache lines to mitigate the probability of uncorrectable errors due to multi-bit upsets (MBUs) in caches with single-error-correction capabilities [31]. In our work we do not consider MBUs since technologies used in critical real-time systems are intended to suffer sufficiently low MBU rates. In any case, this concern is orthogonal to our work. Coarser-grain solutions such as lockstep execution are also common in critical real-time systems [19], [20]. However, those designs often combine also lockstep execution with ECC protection in caches, as in the case of the LEON3FT processor for the space domain [12], thus being orthogonal to our approach.

VI. Conclusions

Emerging real-time applications require increased performance in embedded systems, but also enough reliability levels for specific domains. Write-back L1 caches can help increase performance of such multi-core systems, but dirty data requires additional error correction. Unfortunately, implementing ECC, such as SECDED, increases the end-to-end latency to fetch and correct data. This can result in significant performance degradation due to data dependencies between loads that hit in the L1 cache and the following (consumer) instructions. We propose a novel approach to mitigate this issue, called Look-Ahead Error-Correction (LAEC) that anticipates data loading by one cycle whenever possible to avoid potential stalls. Our results show that our technique improves performance by 6%-13% w.r.t. existing solutions, and is only within 4% of the ideal case where no ECC is needed. Our proposal not only has low execution time overhead but also low design complexity and hardware cost since no costly instruction flush and state recovery is needed.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773) and the HiPEAC Network of Excellence. Pedro Benedicte and Jaume Abella have been partially supported by the MINECO under FPU15/01394 grant and Ramon y Cajal postdoctoral fellowship number RYC-2013-14717 respectively.

References

[1] RENESAS R-Car H3. https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.
[2] A. Agah et al. Tertiary-Tree 12-GHz 32-bit Adder in 65nm Technology. In *ISCAS*, 2007.
[3] J. Abella et al. Low vccmin fault-tolerant cache with highly predictable performance. In *MICRO*, 2009.
[4] J. Abella et al. RVC: A mechanism for time-analyzable real-time processors with faulty caches. In *HiPEAC conference*, 2011.
[5] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.
[6] ARM. ARM Cortex-M7 processor. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B_cortex_m7_trm.pdf.
[7] ARM. ARM Cortex R5 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0460d/DDI0460D_cortex_r5_r1p2_trm.pdf.
[8] ARM. ARM expects vehicle compute performance to increase 100x in next decade. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php, 2015.
[9] P. Benedicte et al. HWP: hardware support to reconcile cache energy, complexity, performance and WCET estimates in multicore real-time systems. In *ECRTS*, 2018.
[10] C.L. Chen and M.Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state of the art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.
[11] Cobham Gaisler. Flight Software Workshop 2017. http://flightsoftware.jhuapl.edu/files/2017/Day-3/02-Hellstrom-Cobham-HiRel.pdf.
[12] Cobham Gaisler. *GR712RC. Dual-Core LEON3FT SPARC V8 Processor. User's Manual*, 2018.
[13] D. Strukov. The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories. In *ACSSC*, 2006.
[14] D. Dasari et al. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, 2011.
[15] M. Shevgoor et al. Efficiently prefetching complex address patterns. In *MICRO*, pages 141–152, Dec 2015.
[16] Michael Paulitsch et al. Mixed-criticality embedded systems - A balance ensuring partitioning and performance. In *Euromicro DSD*, 2015.
[17] N. Muralimanohar et al. CACTI 6.0: A tool to understand large caches. In *HP Tech Report HPL-2009-85*, 2009.
[18] H. Duwe et al. Correction prediction: Reducing error correction latency for on-chip memories . In *HPCA*, 2015.
[19] IBM. *PowerPC 750GX Lockstep Facility. Application note*, 2008.
[20] Infineon. AURIX - TriCore datasheet, 2012.
[21] International Standards Organization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[22] C.-K. Koh et al. Tolerating process variations in large, set-associative caches: The buddy cache. *ACM Trans. Archit. Code Optim.*, 6(2), 2009.
[23] I. Koren and Z. Koren. Defect tolerance in vlsi circuits: techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838, 1998.
[24] B. Maric et al. Efficient cache architectures for reliable hybrid voltage operation using EDC codes. In *DATE*, 2013.
[25] B. Maric et al. Analyzing the efficiency of L1 caches for reliable hybrid-voltage operation using EDC codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2211–2215, 2014.
[26] B. Maric et al. Hybrid cache designs for reliable hybrid high and ultra-low voltage operation. *TODAES*, 20(1), November 2014.
[27] C. McNairy and J. Mayfield. Montecito error protection and mitigation. In *HPCRI*, 2005.
[28] J. Poovey. *Characterization of the EEMBC Benchmark Suite*, 2007.
[29] Freescale Semiconductor. MPC8548E PowerQUICC III. http://cache.freescale.com/files/32bit/doc/data_sheet/MPC8548EEC.pdf.
[30] SoCLib. The soclib project. http://www.soclib.fr/trac/dev.
[31] X. Vera et al. Reducing soft error vulnerability of data caches. In *SELSE Workshop*, 2007.
[32] C. Wilkerson et al. Trading off cache capacity for reliability to enable low voltage operation. In *ISCA*, 2008.
[33] C. Wilkerson et al. Reducing cache power with low-cost, multi-bit error-correcting codes. In *ISCA*, 2010.