RVC: A Mechanism for Time-Analyzable Real-Time Processors with Faulty Caches

Jaume Abella[1], Eduardo Quiñones[1], Francisco J. Cazorla[1,2], Yanos Sazeides[3], Mateo Valero[1,4]
[1]Barcelona Supercomputing Center (BSC-CNS)
[2]Instituto de Investigación en Inteligencia Artificial (IIIA-CSIC)
[3]University of Cyprus (UCY)
[4]Universitat Politecnica de Catalunya (UPC)
jaume.abella@bsc.es, eduardo.quinones@bsc.es, francisco.cazorla@bsc.es, yanos@cs.ucy.ac.cy, mateo@ac.upc.edu

ABSTRACT

Geometry scaling due to technology evolution as well as *Vcc* scaling lead to failures in large SRAM arrays such as caches. Faulty bits can be tolerated from the average performance perspective, but make critical real-time embedded systems non time-analyzable or worst-case execution time (WCET) estimations unacceptably large.

This paper proposes a mechanism to tolerate faulty bits in caches while still providing safe and tight WCET. Our solution is based on adapting structures such as the victim cache, cache eviction buffers or miss state handle registers to serve as replacement for faulty cache storage. We show how modest modifications in the hardware help providing safe and tight WCET on the face of permanent faulty bits with negligible impact in power and performance.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and reliability—*performance analysis and design aids*

General Terms

Performance, Reliability

Keywords

Real-time, Time Analysis, Embedded, Cache, Faults

1. INTRODUCTION

Technology evolution leads to smaller geometries of devices and larger number of transistors in the chip. Process variations [25] are still an issue for processor design, and even if absolute variations decrease, their relative impact increases. Thus, transistors and wires are more likely to misbehave due to variations. On the other hand, limited power budget requires decreasing *Vcc* to keep power and temperature at bearable levels. Lowering *Vcc* exacerbates the impact of process variations in timing because cycle time guardbands must be increased significantly to keep reliability levels constant across generations. However, those large guardbands overwhelm the potential benefits of technology scaling. Hence, some increase in the number of faults must be expected in future technology generations.

Large cache memories occupy a vast area of chips because they provide increased performance at the expense of low power. Most of the cache area is devoted to SRAM cells storing contents. SRAM cells are particularly error-prone at low *Vcc* operation since process variations and noise jeopardize their data retention capabilities [3]. Therefore, SRAM cells require large design effort to keep error rates low. Last level caches (L2 or L3 caches depending on the processor) may lay in a different *Vcc* domain to that of the rest of the chip. Those caches are operated in such a way that retain contents at ultra-low *Vcc* and their voltage is raised in the proper banks when data are accessed. Typically, *Vcc* to serve accesses is higher than that of the rest of the chip to guarantee correct operation. Unfortunately, this approach cannot be applied to L1 caches that lay in the same voltage island than the cores themselves. Thus, even if SRAM cells of L1 caches are larger than those of L2 and L3 caches, their operation *Vcc* is lower and hence, their error rates may be higher than those of caches in independent voltage islands.

Techniques to detect and correct errors in data have been largely studied in the past. Some general methods such as parity and error correction codes (ECC) deal with these issues [5]. In general those methods are effective for soft errors, but are not used to correct errors due to hard faults because they have an impact in performance and reliability. For instance, if a cache line is extended with single error correction double error detection (SECDED) codes and has a faulty bit, each read operation to that particular cache line requires reading the data and correcting it before providing the data to the core, thus impacting latency. Moreover, any soft error in that cache line cannot be corrected by means of SECDED. Thus, such cache line lacks of error correction capabilities for transient errors. In general, those issues are avoided by disabling faulty cache lines. By disabling few cache lines average performance experiences negligible degradation and reliability in the chip is not jeopardized [14, 19, 22].

Performance requirements and energy constraints in real-time systems pushes for processors with cache memories and multiple cores [15]. While disabling faulty storage has been shown to be an effective solution in terms of average performance [1, 21, 26], such a solution may have a large impact in worst-case execution time (WCET) estimations in real-time environments. Given the fact that an accurate analysis of caches is key for tight and safe WCET estimation of real-time applications, disabling cache lines makes WCET analysis inaccurate or highly pessimistic, thus leading to unpredictable or unacceptably large WCET estimations.

Using redundant cache lines to replace faulty ones has been widely used to increase yield, but those approaches require setting up fuses, which are expensive in terms of area [7, 13]. Moreover, re-routing signals to the spare cache lines has an impact in latency that can be paid in non-latency-critical caches such as L2 and L3 ones, but such latency cannot be afforded in performance-critical caches such as L1 ones. Thus, smart approaches based on taking advantage of existing hardware must be exploited to keep embedded processors analyzable to provide safe and tight WCET estimations.

In this paper we propose a mechanism for replacing faulty cache lines with extra cache lines that are added and reserved in cache-assist structures such as victim caches, miss state handle registers, eviction buffers and write combining buffers in L1 cache memories. We show how this cache-like blocks can be extended to replace faulty cache lines, how L1 cache replacement information must be updated to avoid altering L1 cache behavior, and how replament cache lines can be taken into account by the WCET tool. Our approach lies on existing error detection and correction techniques and provides reconfiguration capabilities to keep execution time predictable. We focus on L1 caches because they occupy most of the chip area (excluding L2 and L3, which may use spares) and hence, they are the dominating component in terms of permanent errors. Reconfiguration capabilities for further blocks remain as part of our future work. The contributions of this paper are as follows:

1. The Reliable Victim Cache (RVC). The RVC is a particular implementation of a conventional victim cache that allows replacing faulty cache lines in L1 cache with supplementary cache lines available in the victim cache. We also show that RVC behavior can be also emulated with other hardware such as miss state handle, eviction and write combining buffers.

2. Operation mechanisms to guarantee correct behavior of cache systems complemented with the RVC. This is particularly important to keep replacement information in cache as if there were not faulty cache lines. Otherwise WCET tools require further complexity to analyze several potential states of the replacement information.

3. A detailed evaluation of the impact in performance (average and WCET), power and area of the RVC. Results show that by paying a modest 1% area and energy overhead WCET is not increased at all despite cache faults.

The rest of the paper is organized as follows. Section 2 presents the RVC, our approach to provide time predictability in faulty caches. Alternative implementations of the RVC are detailed in Section 3. Section 4 presents performance (average and WCET), power and area results for the RVC. Section 5 discusses related work. Finally, Section 6 concludes this paper.

2. THE RELIABLE VICTIM CACHE

Hardware may suffer permanent and intermittent faults due to many reasons. One of the most typical ones is the fact that devices (metal stripes, polysilicon polygons, silicon wells, etc.) are fabricated assuming a given geometry, but actual physical devices do not match such nominal geometry due to process variations and limited capabilities of the manufacturing tools. Thus, although physical devices resemble their expected shapes, they do not match them exactly. In fact, in some cases they may show mismatches large enough to make them not operate properly. However, in those cases processors can be typically tested during post-silicon stages and discarded due to their faults. Similarly, some particles may reach the surface of the devices affecting their geometry.

Once a given processor passes all tests successfully, it is considered to be a safe chip and can be used to run real-time tasks. Unfortunately, some devices may be almost faulty due to some imperfections and may fail soon during operation when they degrade enough due to their usage. This kind of early faults is typically referred to as latent defects. Although on-line approaches to detect those failures have been proposed, it may be unfeasible to replace those processors, at least immediately. For instance, those chips may be part of a space engine intended to operate for several years with no way to replace faulty chips (e.g., a satellite), or part of an aircraft, which cannot be repaired until landed.

Latent defects are more likely in smaller devices where the relative impact of a small defect is larger. Large cache arrays are designed using rather small transistors and wires to increase their capacity and decrease their power and delay per unit area. However, such small geometry for devices, its high density and their large area makes caches prone to faults. Thus, we can expect most of the faults to arise in cache memories. Fortunately, the purpose of setting up caches is increasing performance, not providing correctness. Thus, some cache space can be removed while still allowing correct execution of tasks and high average performance [14, 19, 22].

While functional correctness may be enough for general-purpose systems, safety critical embedded systems require also timing correctness to execute all tasks timely not missing any deadline. Soft errors may require some overhead to recover, but their transient nature limits their impact, and hence, some slack can be left available in task scheduling to deal with those errors. However, hard faults, and particularly those in caches, may make real-time tasks to violate their deadlines repeatedly. WCET estimation tools rely on a deterministic behavior of caches. However, faults introduce non-determinism (fault number and location is unknown a priori). Thus, it is critical setting up solutions to keep time analyzability in embedded systems despite of faults instead of making extremely pessimistic assumptions (e.g., assume that all cache lines are faulty and data must be retrieved from main memory).

2.1 Impact in WCET of Faults in Cache Memories

A single faulty bit in a cache line may require disabling the whole cache line or at least part of it. Recently, some literature has proposed how to minimize the amount of cache space disabled due to faults [1,21,26]. Anyway, some space will be disabled. For the sake of this discussion we will assume that the whole cache line is disabled (e.g., the faulty bit is part of the tag).

Once a cache line has been disabled due to a faulty bit, we must measure its impact in performance. In general, such impact is relatively small given the fact that caches may have some hundreds or even thousands of cache lines, and hence, on average few hits will become misses due to the faulty cache line. However, in terms of WCET potentially ***all*** the cache accesses may become misses due to a single faulty cache line. Thus, to be safe, we must make such a pessimistic assumption to estimate the WCET. In order to show this effect we illustrate two different ways to proceed in the presence of a faulty cache line.

- **Scenario A**. The cache line is removed from the replacement stack. Let us assume we have an N-way set-associative cache. Potentially, all cache accesses could be mapped to the very same cache set, and potentially they could access each cache line in a round-robin fashion. For instance, in a 4-way cache with LRU (least recently used) replacement we could have 4 lines (D1, D2, D3 and D4) mapping to the same set and being accessed in a round-robin fashion (see Figure 1). If one of the cache lines in the set becomes faulty all accesses will miss in cache because we will be able to fit only 3 cache lines in the set and each access will replace the cache line that will be used immediately. A less pessimistic way to measure the impact in WCET would consist of assuming that we will lose all the hits to the cache set with the highest number of hits, or at least all those hits in the LRU position of the set. However, performing those accurate estimations would require having a full knowledge of the addresses of cache accesses. Anyway, each extra cache miss may have large latency by having to go through some buses (which may be shared with other cores/devices) and accessing some other blocks such as L2/L3 caches and main memory (which again may be shared with other cores/devices).

- **Scenario B**. The cache line is kept in the replacement stack but its contents are not cached. Following the example in scenario A, we could assume that whenever we have to allocate some data into a physically faulty cache line, we simply serve the data to the processor and do not cache the line. In the example before, all accesses to one of the lines (e.g., D4) would miss in cache (see Figure 1). Potentially, such a solution could make all cache accesses miss in cache if all accesses are performed to data fitting into a cache line that is physically mapped into a faulty cache line. A less pessimistic way to perform this estimation would consider that the physical cache line with the largest number of hits always produces misses. Again, the cost of each cache miss can be huge due the long latency to fetch the data as well as the potential contention that the access can experience in shared or highly busy resources.

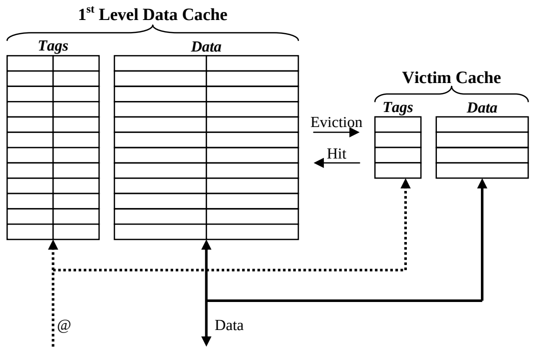
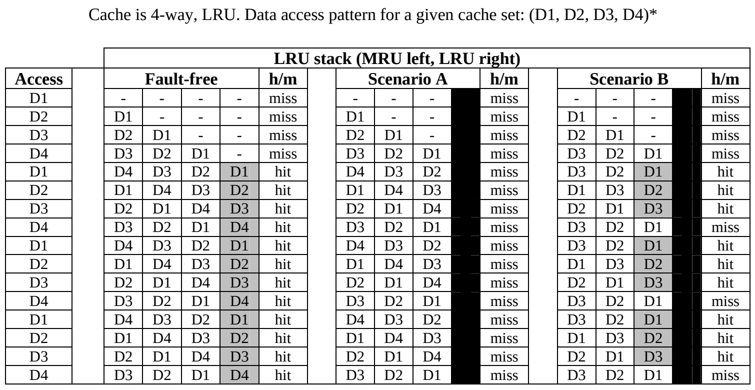


Figure 2: Example of a 2-way 1^{st} level data cache (DL1) with a 4-entry victim cache

2.2 RVC Design

Our first proposal consists of using a slightly-modified victim cache [11] adapted for reliability purposes. The purpose of victim caches is storing those cache lines evicted from first level caches (L1) to allow temporal extra associativity for some cache sets that may need it. Figure 2 shows the schematic of a 2-way L1 data cache and a victim cache. Victim caches can be accessed either (i) in parallel with L1 caches, thus serving data with the same latency as L1 caches, or (ii) sequentially, but only in case of an L1 miss, with a total latency higher than that of an L1 hit.

The basic idea consists of replacing faulty L1 cache lines with extra victim cache lines set up for reliability purposes in such a way that the hit latency to the faulty L1 lines either remains the same (e.g., if the L1 and victim caches are accessed in parallel) or is slightly



Cache is 4-way, LRU. Data access pattern for a given cache set: (D1, D2, D3, D4)*

	LRU stack (MRU left, LRU right)														
Access	Fault-free				h/m	Scenario A				h/m	Scenario B				h/m
D1	-	-	-	-	miss	-	-	-	■	miss	-	-	-	■	miss
D2	D1	-	-	-	miss	D1	-	-	■	miss	D1	-	-	■	miss
D3	D2	D1	-	-	miss	D2	D1	-	■	miss	D2	D1	-	■	miss
D4	D3	D2	D1	-	miss	D3	D2	D1	■	miss	D3	D2	D1	■	miss
D1	D4	D3	D2	D1	hit	D4	D3	D2	■	miss	D3	D2	D1	■	hit
D2	D1	D4	D3	D2	hit	D1	D4	D3	■	miss	D1	D3	D2	■	hit
D3	D2	D1	D4	D3	hit	D2	D1	D4	■	miss	D2	D1	D3	■	hit
D4	D3	D2	D1	D4	hit	D3	D2	D1	■	miss	D3	D2	D1	■	miss
D1	D4	D3	D2	D1	hit	D4	D3	D2	■	miss	D3	D2	D1	■	hit
D2	D1	D4	D3	D2	hit	D1	D4	D3	■	miss	D1	D3	D2	■	hit
D3	D2	D1	D4	D3	hit	D2	D1	D4	■	miss	D2	D1	D3	■	hit
D4	D3	D2	D1	D4	hit	D3	D2	D1	■	miss	D3	D2	D1	■	miss
D1	D4	D3	D2	D1	hit	D4	D3	D2	■	miss	D3	D2	D1	■	hit
D2	D1	D4	D3	D2	hit	D1	D4	D3	■	miss	D1	D3	D2	■	hit
D3	D2	D1	D4	D3	hit	D2	D1	D4	■	miss	D2	D1	D3	■	hit
D4	D3	D2	D1	D4	hit	D3	D2	D1	■	miss	D3	D2	D1	■	miss

Figure 1: Access pattern for 3 different scenarios: a fault-free cache, scenario A and scenario B. Black boxes stand for faulty entries and grey boxes for entries experiencing a hit

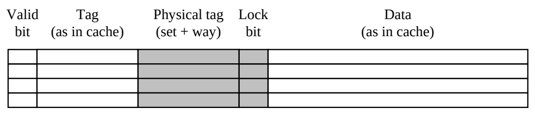


Figure 3: Example of a 4-entry RVC

increased and in a deterministic manner (e.g., L1 and victim caches are accessed sequentially). Typically, victim cache entries consist of a *valid bit*, a *tag* and *data* space to store a full cache line. Our approach requires adding some extra victim cache entries and extending entries with two extra fields: the *lock bit* and the *physical tag* as shown in Figure 3. The *lock bit* indicates whether a victim cache entry has been allocated to replace a faulty cache line. The *physical tag* indicates which cache line this entry replaces (set and way) in case it has been locked.

Similarly to state-of-the-art approaches for fault tolerance [1,2,21,26] we assume that there are mechanisms in place to detect faulty storage and configure the extra fields of the RVC properly. Those mechanisms rely on running some tests at boot time and/or periodically during operation to detect faulty bits and set particular bits indicating whether each storage block is faulty or not[1].

The RVC itself may also experience faults. However, extra entries set up for reliability purposes may serve also to replace faulty victim cache entries. Thus, in case of having a faulty entry in the RVC, there will be one less RVC entry available to be locked. This way extra RVC entries set up for reliability purposes serve as replacement for both faulty L1 and RVC lines. However, when replacing faulty RVC lines they are not locked as they lay in the same structure and replace faulty lines naturally. Note that the number of faulty lines supported in both the L1 and the RVC equals the number of extra entries set up in the RVC. How many extra entries must be set up depends on the cache size and

[1]Note that those bits are hardened or implemented with triple modular redundancy to ensure that they are fault-free.

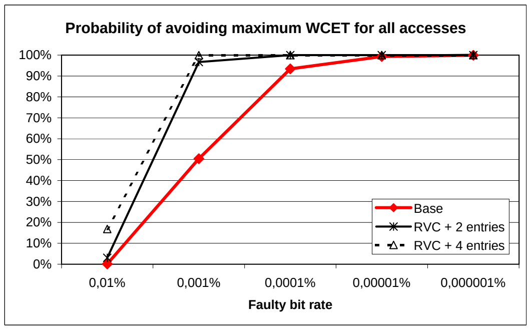


Figure 4: Yield for a 4-way 8KB 64B/line cache with a 4-entry victim cache, 6-entry RVC and 8-entry RVC

faulty bit rate. We illustrate this dependency in Figure 4 for a cache with a 4-entry victim cache and two configurations of the RVC with 2 and 4 extra entries respectively. For instance, for a 0.001% faulty bit rate (1 faulty bit every 100,000 bits) the baseline cache system would have a yield as low as 50.5%. This means that 49.5% of the processors would have faulty bits and WCET estimations would not be valid. Conversely, an RVC with 2 and 4 extra entries increases yield up to 96.7% and 99.9% respectively.

The remaining control bits in cache, such as those devoted to store replacement information (e.g., LRU bits), as well as cache line dirty and valid bits are assumed to be hardened (e.g., using more robust memory cells [9, 10]) as their relative contribution to the cache are and power is negligible.

Next we describe how to manage cache accesses in the presence of faults to keep time analyzability with the help of the RVC. We describe different scenarios with the help of the example in Figure 5.

Cache Hit.

Hits are managed as in a fault-free cache. If there is a hit data are served as usual and replacement information is updated conveniently. From the replacement

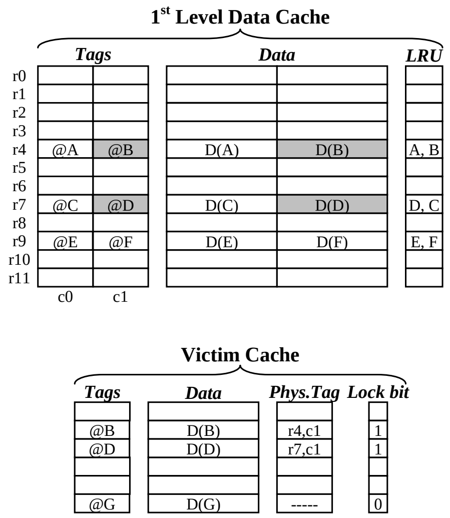


Figure 5: Example of a 2-way DL1 with a RVC. Faulty cache lines are shown in gray

policy point of view faulty cache lines are considered in the same manner as fault-free ones. For instance, this would be the case if we accessed *@A* in Figure 5.

Cache Miss (Miss also in a fault-free cache) - Replace fault-free cache line.

Whenever there is a miss in a fault-free cache, data must be retrieved from upper memory levels. In the case of a faulty cache, whenever there is a miss we do not know whether the miss was caused by a faulty cache line or not. Thus, in terms of WCET computation each miss may have been caused by the faulty cache line.

If we use the RVC data will not be present in the RVC entries devoted to replace faulty cache lines. This is so because the access would have been a miss even in a fault-free cache. In this case data are retrieved from its location (upper memory levels or unlocked RVC entries). There are two different scenarios depending on whether the cache line chosen for replacement in cache is faulty or not. If the cache line is fault-free, the way to proceed is as follows:

(i) Cache access happens as usual. A miss is detected. E.g., we access *@H*, which is mapped in set *r7*.

(ii) The access proceeds to the RVC, either in parallel or sequentially. It misses in the locked entries because in the baseline the access was a miss. We have the physical tag of the cache line being replaced in cache (set and way) (<*r7,c0*>) and will compare it against the physical tags in the RVC. There will not be any match in any locked entry because the cache line chosen to store the data is fault-free, and hence, it has no counterpart in the RVC.

(iii) Data will be requested to upper memory levels or unlocked RVC entries as in the baseline.

(iv) Whenever data are received, they are stored in the corresponding cache line, but not in the RVC. Thus, set *r7* will hold *@H* and *@D* (*@D* is in a faulty line).

(v) In parallel, the evicted cache line (*@C*) is moved to an unlocked entry of the RVC. Potentially it could evict *@G* from the RVC, however, *@B* and *@D* could not be evicted as they are stored in locked RVC entries.

(vi) Cache updates its replacement information. Thus, the LRU stack will be {*H,D*}.

Cache Miss (Miss also in a fault-free cache) - Replace faulty cache line.

Conversely, if the cache line where data are to be allocated is faulty, the way to proceed is somewhat different:

(i) Cache access happens as usual. A miss is detected. E.g., we access *@I*, which is mapped in set *r4*.

(ii) The access proceeds to the RVC, either in parallel or sequentially. It misses in the locked entries because in the baseline the access was a miss (there is no tag match). We have the physical tag of the cache line being replaced in cache (set and way) (<*r4,c1*>) and will compare it against the physical tags in the RVC. There will be a match in this physical tag for exactly one of the locked entries (the one storing *@B*) because data are to be allocated in a physically faulty cache line.

(iii) Data will be requested to upper memory levels or unlocked RVC entries as in the baseline.

(iv) The RVC locked entry mapping the faulty cache line (*@B*) is unlocked and becomes the most recently used RVC entry.

(v) Whenever data are received (*@I*), they are stored in a RVC line, which will be selected according to the replacement policy of the RVC. Such RVC line will be updated with the new tag and data (*@I* and *D(I)*), it will be locked, and its physical tag will be set to the one of the faulty L1 cache line it maps (<*r4,c1*>). Note that if the data requested was in an unlocked RVC entry, we should only update the physical tag and lock fields of the two RVC entries involved, thus avoiding unnecessary data movements.

(vi) Cache updates its replacement information assuming that the data have been allocated in the faulty cache line that was chosen for replacement. Thus, the LRU stack will be {*I,A*}.

Cache Miss (Hit in a fault-free cache).

Whenever there is a hit in a fault-free cache, data can be served immediately. In the case of a faulty cache, if the access hits a fault-free cache line data are served as in the baseline case as explained before. However, it may be the case that there is a cache miss because the cache line where the data would have been hold was faulty (e.g., accessing $@B$ in the example). Data will be present in a locked entry of the RVC. The way to proceed is as follows:

(i) Cache access happens as usual (e.g., we access $@B$). A miss is detected because the cache line is marked as faulty.

(ii) The access proceeds to the RVC, either in parallel or sequentially. It must hit in one of the locked entries because in the baseline the access would have been a hit. Thus, the RVC serves the data. Since the RVC has a physical identifier of the faulty cache line (set and way), it provides this information to the cache along with the data.

(iii) The cache updates its replacement information assuming that there has been a hit in the cache line indicated by the RVC (the faulty cache line). Thus, the LRU stack will be $\{B,A\}$.

As shown, at the end of the access data are served and the replacement stack of the cache holds the same information as in the baseline case. Thus, we can guarantee the same behavior with one exception: all misses in a set with at least a faulty cache line (those that would have happened in the baseline and those caused by faulty cache lines) observe a deterministic latency increase[2] to access the RVC before proceeding to the upper memory levels.

2.3 Impact in WCET

In this section we describe how to account accurately for the impact in the WCET of failures, both with and without the RVC.

L1 and RVC Accessed in Parallel.

If a given L1 cache line is faulty, the RVC provides the data that such L1 cache line should hold if it was fault-free. If both the L1 cache and the RVC are accessed in parallel, the access latency is the same independently of which of both structures holds the data. Hence, our approach does not impact the WCET with respect to a fault-free cache system with a victim cache.

L1 and RVC Accessed Sequentially.

If the cache memory and the victim cache are accessed sequentially, data that should be hold in a faulty L1 cache line is available in the RVC. Thus, the overall access latency is increased by the latency of the RVC. Note that knowing such latency is critical for WCET estimation tools, which build their estimations assuming deterministic behavior of processor components.

In general, in a processor free of timing anomalies [20] like those used for safety critical embedded systems, an access experiencing a hit in L1 has the following WCET:

$$WCET = hit\_latency(L1) \tag{1}$$

[2]Latency increases only if the L1 cache and the RVC are accessed sequentially. If they are accessed in parallel there is no such latency increase.

Outcome L1/VC	**Baseline Fault-free**	**Baseline Faulty**	**RVC Fault-free**	**RVC Faulty**
hit/-	2	104	2	4
miss/hit	4	104	4	4
miss/miss	104	104	104	104

Table 1: WCET latency for different types of access for the baseline and the RVC design in fault-free and faulty environments

Conversely, if the access experiences a miss in L1 but hits in the victim cache (VC) in the baseline or the RVC in our proposal, its WCET is as follows:

$$\begin{aligned} WCET = {} & miss\_latency(L1) + \\ & + hit\_latency(RVC) \end{aligned} \tag{2}$$

Finally, if the access experiences a miss both in L1 and the VC (or RVC), its WCET can be expressed as follows:

$$\begin{aligned} WCET = {} & miss\_latency(L1) + \\ & + hit\_latency(RVC) + mem\_latency \end{aligned} \tag{3}$$

where *mem_latency* stands for the corresponding latency for the next levels of the memory hierarchy beyond the VC/RVC. Thus, it includes other cache levels and DRAM memory.

In case of having faulty lines, accesses that would have experienced a hit in a fault-free cache or a fault-free VC may observe a significant increase in their latency. In particular, those accesses should have had the WCET indicated in equation 1 or equation 2 but, instead, they have the WCET indicated in equation 3 as we cannot determine whether their data will be available in L1 or the VC due to the faulty cache lines. Thus, given that a significant fraction of instructions are memory accesses and the latency to access main memory is much higher than L1/VC latency, we can expect a huge impact on the overall WCET due to faulty L1/VC cache lines.

If we use the RVC, we can guarantee that all cache accesses that would have hit in L1/VC in a fault-free cache system, will hit at least in the RVC. Thus, only those accesses with the WCET shown in equation 1 will observe a slight increase because they WCET will be that in equation 2.

Example.

In order to illustrate the advantage of the RVC we use the example of a cache system where hit and miss latencies for L1 and VC/RVC is 2 cycles, and memory latency is 100 cycles. WCET latencies for each type of access in a fault-free baseline are shown in Table 1.

It can be observed that the RVC design only has a small impact in the WCET of those accesses that would have hit in L1 in a fault-free cache, whereas the WCET does not increase otherwise. Conversely, if no RVC is in place, WCET latency can be as high as the latency to access main memory because any access can become a miss (see scenario A in Figure 1).

3. ALTERNATIVE RVC IMPLEMENTATIONS

Current processors include structures to allow store operations and dirty line evictions be performed in background without stalling the pipeline. For instance, write-back caches use eviction buffers (EB for short) to hold those dirty lines evicted from cache until they can be updated in other cache levels.

Cache accesses check both the cache memory and the EB either in parallel or sequentially (as for the RVC) to determine whether there is a hit or a miss. Thus, instead of using the victim cache, we can use the EB to implement the reliability enhancements required for a tight WCET computation. The implementation is analogous to that of the RVC since we have to add few extra entries and the same fields to each entry as in the case of the RVC. Those entries are operated also as in the RVC.

Those extra EB entries are not available when estimating the WCET. Therefore, if the cache memory and the EB are accessed in parallel (the most typical implementation), the WCET is not impacted at all. Instead, if the cache and the EB are accessed sequentially, the impact in the WCET is analogous to that for the sequentially accessed RVC increasing slightly the latency for those cache hits that should had happened in faulty cache lines as described in previous section.

Other cache implementations use write-through policy instead of write-back. In those cases a write combining buffer (WCB for short) is used instead of an EB. Cache lines cannot be dirty because they are immediately updated in the corresponding cache levels, so there is no need for an EB. However, the WCB allows sending data to the other cache levels when the bus is available without stalling the pipeline and combining different store instructions whose data map into the same cache line. When accessing the cache memory, both the cache and the WCB are accessed either in parallel or sequentially as it is the case for the EB. Thus, we can implement the reliability enhancements required for tight WCET computation in the WCB in the same way as in the EB and the RVC. By allocating few extra WCB entries for replacing faulty cache lines we avoid undesirable side effects in the WCET computation. Thus, WCET remains the same if the cache memory and its WCB are accessed in parallel, and is slightly increased in some particular cases as for the RVC and the EB.

Load misses and store misses[3] are kept in miss state handle registers (MSHR for short). Those registers group requests to the same cache line to avoid issuing redundant requests to the upper level of the memory hierarchy, and store the data when they arrive because the bus is typically narrower than a cache line. Once the cache line has been filled, it is updated in the L1 cache to avoid inconsistencies. The way to manage the MSHR is analogous to that of the RVC, EB and WCB, because it can be accessed in parallel or sequentially with L1. Again, the MSHR can be extended as those other buffers with identical impact in WCET.

[3]Under write allocate policy

Although solutions presented so far focus on L1 caches, they can be used for any cache in the memory hierarchy implementing victim caches, miss state handle buffers, eviction buffers or write combining buffers. For instance, EB or similar buffers should be used at any cache level to avoid locking cache memories during long time periods.

4. EVALUATION

This section presents the evaluation framework and results in terms of WCET, performance, power and area for the RVC design when the L1 data cache is faulty. Results for the L1 instruction cache are not included due to lack of space and because they do not provide further insights.

4.1 Evaluation Framework

Impact on WCET has been studied for a set of scenarios with different degree of dependency on each level of the data cache hierarchy. Thus, we have considered synthetic benchmarks with different breakdowns of the number of instructions in the execution path considered for the WCET estimation. Instructions execute serially and do not overlap their execution at all. Instructions are classified into several groups (i) not accessing cache, (ii) accessing cache with guaranteed hit latency in the L1 cache, (iii) accessing cache with guaranteed hit latency in the RVC, MSHR, EB or WCB, and (iv) accessing cache with no hit guarantee (hence, miss latency is assumed). Such a study helps understand the potential impact in WCET that each scenario may experience.

Although the extra VC/MSHR/EB/WCB entries are set up for reliability purposes, they can be used to increase performance if the cache system is fault-free. This fact is particularly relevant for non real-time tasks, which do not care about WCET but about average performance. In order to evaluate such performance gains we use an in-house cycle-accurate, trace-driven simulator compatible with PowerPC and Alpha ISA. The simulator models an in-order core with dual-issue pipeline with a fetch bandwidth of 2 instructions. The size of the instruction buffer is 16, while the instruction window is 8. Branch prediction is enabled for non real-time tasks and uses a Gshare predictor with 256 entries. Store operations do not block the pipeline and they access the cache directly, unless the write buffer is full.

The core has a private first level of cache with separated data and instruction caches. The Instruction and Data L1 caches are 4-way 16KB each. A MSHR with 4 entries is considered in the baseline case. It is extended with 2 extra entries to implement our approach. Since the remaining levels in the memory hierarchy are not relevant for our approach, we simply assume a latency of 100 cycles to access main memory. If such latency is smaller (e.g., a L2 cache is in place) benefits will be lower. Conversely, if memory latency is higher benefits will be higher. Regarding the workload, we use the SPECCPU2000 benchmark suite, which is representative of non real-time workloads [23].

Power and area results have been obtained by means of the CACTI 6.5 tool [16,24]. CACTI is a delay, power

Name	Fixed	L1	VC/MSHR/...	MEM
L1a	0.9	0.05	0.04	0.01
L1b	0.8	0.15	0.04	0.01
L1c	0.7	0.25	0.04	0.01
L1d	0.6	0.35	0.04	0.01
VMEWa	0.9	0.05	0.04	0.01
VMEWb	0.8	0.05	0.14	0.01
VMEWc	0.7	0.05	0.24	0.01
VMEWd	0.6	0.05	0.34	0.01
MEMa	0.9	0.05	0.04	0.01
MEMb	0.8	0.05	0.04	0.11
MEMc	0.7	0.05	0.04	0.21
MEMd	0.6	0.05	0.04	0.31

Table 2: Synthetic benchmark instruction breakdown

and area model for cache memories developed at the HP Labs. Results have been gathered for 32nm process technology, assuming low power design style, 330K temperature and 0.6V operating voltage. We have validated that those trends observed hold for other technology nodes, design styles, and temperature and voltage values.

4.2 WCET

As explained above, several synthetic benchmarks are considered with different instruction breakdowns. Those configurations are depicted in Table 2. Fixed stands for the fraction of instructions not accessing cache. For the sake of simplicity we assume a latency of 1 cycle for those instructions. L1, VC/MSHR/... and MEM correspond to the fraction of instructions with guaranteed L1 hit latency, VC/MSHR/EB/WCB hit latency and memory latency respectively. We consider both parallel and sequential L1 and VC/MSHR/EB/WCB access. In the case of the parallel access, the fault-free baseline has a L1 and VC/MSHR/EB/WCB hit latency of 2 cycles. In case of miss it takes 102 cycles fetching data from memory. Our approach has exactly the same latency as the baseline despite of the faulty cache lines. Conversely, the faulty baseline must assume always a latency of 102 cycles for each access as it cannot determine which cache line is faulty, and hence, which accesses are guaranteed to hit in any cache structure. In the case of the sequential access we use those latencies in Table 1.

As shown, we consider three sets of configurations. The number of L1 hits, VC/MSHR/EB/WCB hits and memory accesses is increased in the first, second and third set respectively.

Figure 6 shows the relative WCET for the parallel access scenario with respect to the fault-free baseline. Results are depicted for a faulty baseline system and a faulty system with our approach incorporated. Note that a fault-free system with our approach has exactly the same latency as the fault-free baseline system, so we do not include those results in the figure. As shown, the WCET grows dramatically for the faulty baseline cache system. The only exceptions are MEM configurations because a large fraction of the instructions experiences high latencies to access main memory, and therefore, by increasing the latency of those few instructions that would have hit in cache in a fault-free system, the relative latency is increased much less than in the other configurations. On the other hand, the WCET for our approach is exactly the same as for the fault-free baseline since our approach guarantees that all access have exactly the same latency as in the baseline.

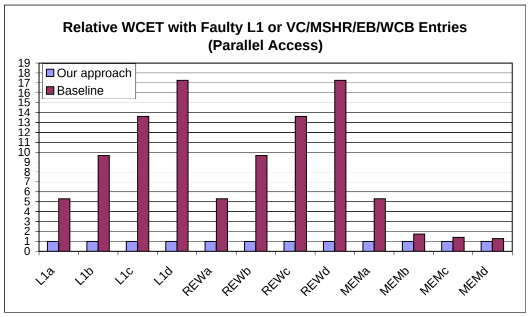


Figure 6: Relative WCET for parallel access to the L1 cache and the VC/MSHR/EB/WCB when few lines are faulty

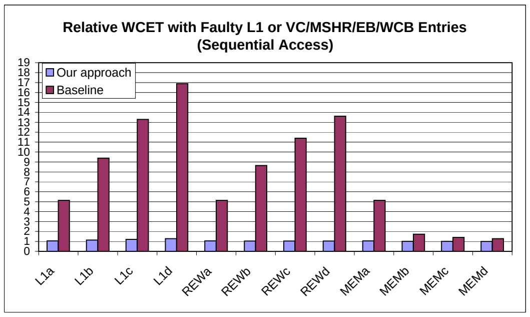


Figure 7: Relative WCET for sequential access to the L1 cache and the VC/MSHR/EB/WCB when few lines are faulty

Figure 7 shows the relative WCET for the sequential access scenario with respect to the fault-free baseline. Trends are roughly the same as for the parallel access scenario. The only difference is the fact that our approach experiences slight WCET increases due to the increased latency of those instructions that hit in L1 (from 2 to 4 cycles). Such increase can be observed mainly in those configurations where a large fraction of instructions were estimated to hit in L1. For instance, the worst configuration is *L1d* where the WCET increases by 1.3X for our approach. Nevertheless, such increase is largely below that of the faulty baseline 16.9X.

Thus, as long as some instructions hit in L1 or the VC/MSHR/EB/WCB in the WCET estimation, our approach provides benefits. As shown, those benefits are huge in general.

4.3 Performance

Although set up for reliability purposes, the extra VC/MSHR/EB/WCB entries can be either turned off or used for increased performance during fault-free operation. If used for performance purposes, results show a modest average speedup of 0.5% (up to 1.8% for `mcf`).

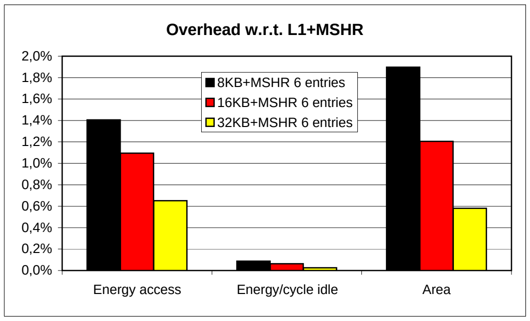


Figure 8: Overhead of the 2 extra MSHR entries with respect to the L1+MSHR cache system (L1 is 4-way 64byte/line 1 R/W port, MSHR has 4 entries)

This makes sense because otherwise would mean that the MSHR was underdesigned. We have also evaluated a 2-entry MSHR, but several programs (`fma3d`, `gzip`, `lucas`, `mcf` and `swim`) experience slowdowns above 5% with respect to the 4-entry MSHR, so a 4-entry MSHR is the best configuration as baseline. Nevertheless, both choices, using the extra entries for performance purposes or keeping them turned off during fault-free operation, are valid and it is up to the designer choosing the most suitable one for the target applications.

4.4 Power and Area

We have studied the impact in power, area and delay of the extra entries in the VC/MSHR/EB/WCB required by our approach. In particular, we have evaluated the impact of adding 2 extra entries in a 4-entry MSHR. As expected, the impact in delay is negligible. Moreover, since the delay of those structures is largely below the cycle time dictated by the L1 cache, there is no impact in terms of cycle time. In terms of power and area, there is an overhead 11% and 9% respectively. However, such overhead is very low in absolute numbers. In order to put those results in perspective, we depict the overhead of those extra MSHR entries with respect to the total power and area of the L1 and the MSHR together since those extra entries deal with faulty cache lines in both structures. Results are shown in Figure 8. As shown, dynamic power overhead ranges between 0.5% and 1.5% for different L1 cache sizes when an access is performed. Idle power overhead is negligible as the extra number of SRAM cells is very small. Finally, area is impacted between 0.5% and 2% mostly because of the routing and peripheral logic of the increased MSHR.

5. RELATED WORK

Several approaches have been proposed to deal with hard errors due to defects and degradation. The most relevant approaches consider alternative memory cell designs which offer higher robustness [9, 10]. However, those memory cells require larger area, and hence, the effective cache space reduces for a given area budget. Moreover, even if those memory cells are employed large cache arrays are still the most error-prone blocks in processors.

Error correction and detection schemes for memory blocks have been widely deployed in the past. The most popular approaches are parity, SECDED (single error correction double error detection) and DECTED (double error correction triple error detection) [5]. Typically, such techniques are used for soft error detection and correction. Data is served when read and, in parallel, error detection and correction is performed. If errors are detected, then some actions are taken to stop execution and provide the correct data. This process is expensive, but given that soft errors are seldom, its impact in performance is irrelevant. However, if those techniques (SECDED and DECTED) are set up to deal with hard errors, data cannot be served until they are corrected. Thus, there is an impact in the data access latency. Moreover, some degree of protection for other errors is sacrificed permanently. For instance, a soft error cannot be corrected if correction capabilites are used to deal with hard errors. Thus, in general, faulty storage is disabled or replaced.

Some approaches have focused on disabling faulty storage to keep operating despite of the failures. In some cases spare storage is set up to replace faulty one [7, 13]. Unfortunately, such storage occupies large area for the fuses required to configure it. Additionally, hardware re-routing signals to the spare storage and wire delay make spares affordable only for slow structures such as L2 and L3 caches, where latency is not critical. Some recent work [6] uses a somewhat similar idea to ours by setting up some redundant storage, which is used to replace the slowest cache entries to improve binning. While this extra storage can be used also to replace faulty storage as our approach does, the authors do not show the implications of their approach on the WCET or how to deal with faults on the redundant storage itself. Other approaches require a level of indirection to replace faulty storage [4], which has an impact in terms of WCET and performance even if the system is fault-free. Other techniques have been devised to disable faulty storage without replacement at different granularities [1, 2, 12, 21, 26]. While purely storage disabling techniques have been shown to be very efficient in terms of average power and performance, they do not offer any type of performance guarantee required for WCET estimation.

Some works have addressed issues related with time analyzability in systems with cache memories assuming fault-free caches during the static analysis [8, 17, 27]. None of these works considers the potential impact of faults in caches. Some recent work has provided means to obtain safe and tight WCET estimation in multicores with shared cache memories by enabling cache partition [18]. This approach follows the same spirit as ours by enabling hardware features to provide time analyzability. However, it does not support faults in caches. To the best of our knowledge there is only one related work providing some time predictability in faulty caches [1]. Unfortunately, this technique provides neither time analyzability nor any performance guarantee, as required by WCET estimation in real-time systems.

6. CONCLUSIONS

Technology scaling and decreased *Vcc* lead to higher SRAM faulty bit rates, especially in cache memories. This issue is particularly relevant for critical real-time embedded systems where unpredictable latency of memory accesses leads to non deterministic systems or unaffordable WCET estimates.

In this paper we propose a mechanism to provide tight and safe WCET estimations in the presence of faulty caches. To the best of our knowledge it is the first approach dealing with this issue. Our approach introduces small extensions either in the VC, the MSHR, the EB or the WCB to replace faulty entries in both cache memory and this structure itself, while still keeping timing analyzable. We show how the WCET is not increased at all (when cache and VC/MSHR/EB/WCB are accessed in parallel) or is slightly increased (when cache and VC/MSHR/EB/WCB are accessed sequentially) with our approach. Conversely, a faulty baseline cache system may experience a large WCET increase. Our experiments show that the WCET increases by more than 17X for some scenarios for the baseline, whereas our approach keeps the WCET within 1.0X and 1.3X that of the fault-free baseline at the expense of 1.1% power and 1.2% area increase at most for the cache system.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Education and Science under grant TIN2007-60625 and Beatriu Pinós 2009 BP-B 00260.

7. REFERENCES

[1] J. Abella et al. Low vccmin fault-tolerant cache with highly predictable performance. In *MICRO*, 2009.

[2] J. Abella et al. The split register file. In *DATE*, 2010.

[3] A.J. Bhavnagarwala, T. Xinghai, and J.D. Meindl. The impact of intrinsic device fluctuations on CMOS SRAM cell stability. *IEEE Journal of Solid-State Circuits*, 36(4):658–665, 2001.

[4] F.A. Bower et al. Tolerating hard faults in microprocessor array structures. In *DSN*, 2004.

[5] 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.

[6] A. Das et al. Evaluating the effects of cache redundancy on profit. In *MICRO*, 2008.

[7] T.P. Haraszti. A novel associative approach for fault-tolerant MOS RAMs. *IEEE Journal of Solid-State Circuits*, 17(3):539–546, 1982.

[8] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[9] M. Ishida et al. A novel 6T-SRAM cell technology designed with rectangular patterns scalable beyond 0.18 μm generation and desirable for ultra high speed operation. In *IEDM*, 1998.

[10] S.K. Jain and P. Agarwal. A low leakage and SNM free SRAM cell design in deep sub micron CMOS technology. In *VLSID*, 2006.

[11] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a), 1990.

[12] C.-K. Koh et al. Tolerating process variations in large, set-associative caches: The buddy cache. *ACM Trans. Archit. Code Optim.*, 6(2):1–34, 2009.

[13] I. Koren and Z. Koren. Defect tolerance in vlsi circuits: techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838, 1998.

[14] H. Lee, S. Cho, and B.R. Childers. Performance of graceful degradation for cache faults. In *ISVLSI*, 2007.

[15] MERASA. Multi-core execution of hard-real-time applications supporting analysability. In *FP7 project. http://www.merasa.org*, 2007-2010.

[16] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. *HP Tech Report HPL-2009-85*, 2009.

[17] F. Nemer et al. Inter-task WCET computation for a-way instruction caches. In *SIES*, 2008.

[18] M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[19] A.F. Pour and M.D. Hill. Performance implications of tolerating cache faults. *IEEE Trans. Comput.*, 42(3):257–267, 1993.

[20] J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

[21] D. Roberts, N.S. Kim, and T. Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. In *DSD*, 2007.

[22] G.S. Sohi. Cache memory organization to enhance the yield of high performance vlsi processors. *IEEE Trans. Comput.*, 38(4):484–492, 1989.

[23] SPEC. *Standard Performance Evaluation Corporation. SPEC CPU benchmark suite.* http://specbench.org/osg/cpu2000, 2000.

[24] S. Thoziyoor, N. Muralimanohar, and N.P. Jouppi. CACTI 5.0. *HP Tech Report HPL-2007-167*, 2007.

[25] O.S. Unsal et al. Impact of parameter variations on circuits and microarchitecture. *IEEE Micro*, 26(6):30–39, 2006.

[26] C. Wilkerson et al. Trading off cache capacity for reliability to enable low voltage operation. In *ISCA*, 2008.

[27] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS*, 2008.