Cache Design for Mixed Criticality Real-Time Systems

N G Chetan Kumar, Sudhanshu Vyas, Ron K. Cytron*, Christopher D. Gill*, Joseph Zambreno and Phillip H. Jones
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA
Email: {ckng, spvyas, zambreno, phjones}@iastate.edu
* Department of Computer Science and Engineering, Washington University, St. Louis, MO, USA
Email: {cytron, cdgill}@cse.wustl.edu

***Abstract*—Shared caches in mixed criticality systems are a source of interference for safety critical tasks. Shared memory not only leads to worst-case execution time (WCET) pessimism, but also affects the response time of safety critical tasks. In this paper, we present a criticality aware cache design which implements a Least Critical (LC) cache replacement policy, where a least recently used non-critical cache line is replaced during a cache miss. The cache acts as a Least Recently Used (LRU) cache if there are no critical lines or if all cache lines are critical in a set. In our design, data within a certain address space is given higher preference in the cache. These critical address spaces are configured using critical address range (CAR) registers. The new cache design was implemented in a Leon3 processor core, a 32bit processor compliant with the SPARC V8 architecture. Experimental results are presented that illustrate the impact of the Least Critical cache replacement policy on the response time of critical tasks, and on overall application performance as compared to a conventional LRU cache policy.**

I. Introduction

Cache memories greatly improve the overall performance of processors by bridging the increasing gap between processor and memory speed. In real-time systems, it is necessary to accurately estimate the worst-case execution time (WCET) of a task to ensure tasks are completed within certain deadlines. The unpredictable behavior of shared caches complicates WCET analysis [14], which leads to overestimation of WCET and decreases processor utilization. Various techniques such as cache locking and partitioning have been proposed to make shared caches more predictable in real-time systems. Higher predictability is often achieved at the cost of reduced application performance.

In mixed criticality real-time systems, where tasks of different criticalities are executed on the same platform, it is necessary to ensure the timing constraints of critical tasks are met under all conditions, while trying to maximize average processor utilization. To achieve this, we need to mitigate the interference of lower criticality tasks on the timing behavior of higher criticality tasks. Shared caches in mixed criticality systems is one such source of interference that can increase the response time of critical tasks.

In this paper, we present a cache design for mixed criticality real-time systems in which critical task data is least likely to be evicted from cache during a cache miss. We assume data is either critical or non-critical. An extension of the least recently used (LRU) cache replacement policy, called Least Critical (LC), is implemented as a part of our proposed cache design. In our LC cache replacement policy, data from certain address spaces are given preference in the cache. These critical address spaces are defined by a critical address range (CAR), which is configurable during run-time. Our design enables fine grained control over classifying task data as critical, and allows run-time configuration of a critical address space to better manage cache performance.

II. Related Work

In the context of shared caches in real-time systems, various cache locking and partitioning schemes have been proposed to improve predictability and overall performance of real-time tasks. In cache partitioning, a portion of the cache is assigned to a task and the task is restricted to only use that assigned partition. This removes inter-task cache conflicts. Software based partitioning techniques such as [5], [2], [15], [6] require changing from address to cache-line mapping to eliminate inter-task conflicts, which makes it difficult for system-wide application. The use of hardware based techniques [8], [12] is limited by fixed partition sizes and coarse grained configurability, which may reduce cache utilization. Cache locking allows certain lines of the cache to be locked in place, which enables accurate calculation of memory access times. While cache locking [13], [1], [11] provides fine grained control over task data, it will lead to poor utilization when data does not fit in the cache [13]. Dynamic cache locking also increases overhead and can affect overall task performance, if cache lines are locked unnecessarily.

More recently, cache management techniques for mixed criticality real-time systems have been proposed to improve predictability and performance of critical tasks. PRETI, a partitioned real time cache scheme was presented in [7], where a critical task is assigned a private cache space to reduce inter-task conflict. The cache lines not claimed by a task are marked as shared, and can be used by all tasks. [9] proposed a cache management framework for multi-core architectures to provide a deterministic cache hit rate for a set of hot pages used by a task. Cache scheduling and locking techniques to manage shared caches within the MC^2 scheduling framework [10] was presented in [4].

In our proposed cache design, we allow fine grained control over task data by providing a mechanism to store critical data in separate address spaces. This enables better cache utilization as the non-critical cache lines are shared by all tasks. By placing critical task data in separate address ranges, which are given preference in cache, the overhead involved in locking/unlocking individual cache lines is also eliminated. Our design provides the flexibility to change critical address ranges at run-time, which enables applications to better utilize cache.

III. Criticality Aware Cache Design

We present a criticality aware cache design for shared caches in mixed criticality real-time systems to reduce inter-task conflicts and decrease response time of critical tasks. The core of the design is a new cache replacement policy, called Least Critical, which is described in detail next.

A. Least Critical Cache

Our Least Critical cache (LC cache) replacement policy targets set associative shared caches in mixed criticality real-time systems. The LC policy is an extension of a conventional least recently used (LRU) cache. For each cache set, we keep a count of lines which have data from critical address range. We also maintain LRU order for critical and non-critical lines in each cache set. During a cache hit, the LRU order of either critical or non-critical lines in the cache set is modified based on the line being accessed. When there is a cache miss, the line to be replaced is selected based on the following order: 1. Empty cache line. 2. Least recently used non-critical cache line. 3. Least recently used critical cache line, if all the lines in a cache set are critical.

During a cache miss, if the data accessed or evicted is from a critical address range, then the number of critical cache lines in that set is updated. During a cache miss, a critical cache line gets evicted only when all lines in a cache set are critical. The LC cache replacement policy acts as LRU, if all the lines in a cache set are from a critical address range or if there is no critical data in a cache set. A working example of the LC cache policy is shown in Figure 1.

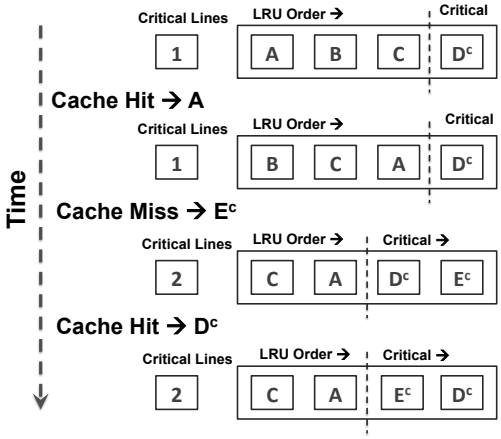


Fig. 1. A working example of our Least Critical cache replacement policy. The LRU order, for both critical and non-critical data, is maintained using a state transition table. ^{C} indicates critical cache lines.

B. Hardware Implementation

Figure 2 depicts a high level block diagram of the LC cache architecture. It is composed of four primary components: 1) CAR Compare, 2) Access History, 3) Tag Compare, and 4) Data Control.

Critical Address Range (CAR) Compare. CAR registers are used to identify critical data based on memory address. An application configures these memory-mapped registers to specify where critical data resides in memory. The architecture supports the use of multiple CAR registers sets, each defines an address space for holding critical data. The memory address is compared with CAR registers during cache access to identify critical cache lines. The implementation of our architecture additionally allows dynamically switching between our LC cache policy and a conventional LRU policy at run-time.

Access History. The LRU order of critical and non-critical lines along with the number of critical lines is maintained as an access history, which is updated on every memory access. In addition to the bits used to store the LRU order for each set, $\log A + 1$ bits are required to track the number critical lines in each set, where A is the cache set associativity.

Tag Compare. Generates cache hit/miss signals by comparing requested memory addresses with tag bits associated with each cache line.

Data Control. Provides an interface to the CPU to read/write data from cache or main memory.

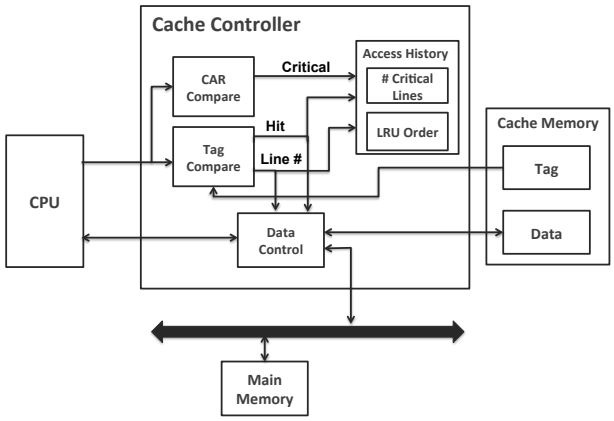


Fig. 2. High level block diagram of the Least Critical (LC) Cache Controller.

The design allows run-time modification of the critical address ranges. Since modifying CAR registers at run-time impacts the coherency of the critical-cache-line count, a mechanism is needed to restore coherency. For maintaining coherency, a *soft-reset* mechanism is used to clear the critical data line count of each cache set to zero. This is accomplished by the application writing to a specific memory mapped register. A *soft-reset* of the cache should be performed before updating CAR registers.

Switching between LC and LRU. The LC cache can be reverted to behaving as LRU by clearing the CAR registers and triggering a *soft-reset* of the critical line counts. After a *soft-reset* of the LC cache, existing critical cache lines default to most recently used non-critical cache lines.

Application-level usage model of LC cache. To make use of the LC cache, the application should tag critical data variables and the compiler should place those variables in a separate section of memory. In GCC, this could be accomplished using the "*section*" attribute, which specifies that a variable resides in a particular section. $Ex.\ int\ cdata\ \_\_attribute\_\_\ ((section("critical")));$

Frequently used memory pages e.g., from the application could also be made critical by configuring CAR registers.

When compared to cache locking, our technique avoids the run-time overhead of locking mechanisms and allows critical data to stay in cache. We also provide graceful degradation when critical data is larger than the cache size, since the cache acts as LRU when all the lines in a set are critical.

IV. Evaluation Methodology

A. Hardware Platform and Configuration

The LC cache replacement policy was evaluated on a XUPV5-LX110T, a Xilinx FPGA development and evaluation platform that features a Virtex-5 FPGA, 256 MB RAM (DDR2), JTAG and UART interfaces. Leon3, a 32bit soft-core processor compliant with the SPARC V8 architecture, was used to implement our cache design. Leon3 features a 7-stage pipeline and separate instruction and data caches. In this paper, we limit the analysis to data cache only. Our cache design was implemented as a L1 data cache in the Leon3 processor running at 33MHz with no memory management unit (MMU). For our evaluation, we used a 4-way set associative data cache of size 4KB with 16 bytes/line. The LRU cache supported by Leon3 was used as the baseline to compare the performance of our LC cache design. A non-intrusive hardware cache profiler was designed to accurately measure the performance of the data cache unobtrusively. The profiler could be configured to measure data cache hits and misses for each task, along with overall application statistics. The profiler sends the data offline to a server through a UART interface.

TABLE I. Characteristics of benchmark programs used to evaluate our cache design.

Task Name	Code Size (bytes)	Data Size (bytes)	Execution Time (ms)[1]
CRC	1216	1048	0.16
FDCT	2940	132	0.49
FIR	572	2948	54.06
Compress	3316	2416	18.52
IPC	1092	256 - 8192	0.11 - 4.87

[1] Execution time for task running alone.

B. Workload and Metrics

To evaluate the performance of our cache design, we used a set of five real-time benchmark programs. The critical task was an inverted pendulum controller (IPC). We varied the resolution of the controller so that its critical data (matrix used in the control computation) ranged from 256 to 8K bytes. Background tasks were drawn from the WCET project [3] and consisted of CRC, FDCT (discrete cosine), FIR (finite impulse response filter), and compress. The characteristics of these programs are shown in Table I. FreeRTOS, an open source kernel designed for embedded real-time systems, was used to run the benchmark applications on Leon3. FreeRTOS was configured to execute a preemptive priority based scheduling algorithm. The cache miss rate of both the critical task and the overall application was measured for LC and LRU cache replacement policies.

V. Results and Analysis

To evaluate the performance of data cache, the benchmark programs were executed using rate monotonic (RM) scheduling. The period of non-critical tasks were kept constant at 200ms and the experiment was conducted for three different critical task periods (50ms, 100ms, 200ms). Figure 3 shows the cache miss rates for our critical task, as the size of its critical data increases. With the LC policy, its references are favored and we generally see a marked improvement over the LRU policy for the critical task. When the size of the critical task's bytes reach the cache size (4K), we see an increase in the critical task miss rate even for the LC replacement policy. This is because the critical tasks' references are due to matrix multiplications, which will incur misses once the cache size is exceeded. Finally at 8K critical data bytes, we have exceeded the size of the 4K-byte cache. Then, LRU and LC are indistinguishable for the critical task.

When using LRU, the miss rate of the critical task increases with its period as shown in Figure 3. This is due to inter-task interference increasing when the critical task is not executed often. In comparison, the LC cache shows a predictable miss rate for the critical task while performing 40% - 70% better than the LRU cache. The LC cache reduces the impact of inter-task conflicts on the critical task by giving preference to that task's critical data.

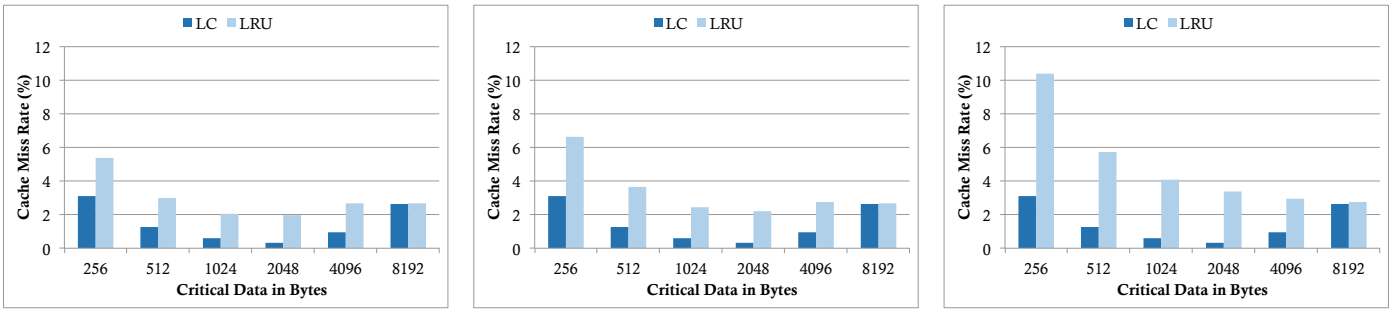
The cache miss rates for the overall application (critical and noncritical tasks) is shown in Figure 4. Overall performance is not adversely affected by LC's favoring the critical task, until we reach the size of the L1 cache at 4K bytes. Comparing across the figures, at 4K, we see reason to favor the critical task, improving its execution time at the expense of the noncritical tasks. However, at 8K, favoring the critical task benefits neither that task nor any other task. LRU would be a better choice at this point.

VI. Conclusion

In this paper, we presented a criticality aware cache design for mixed criticality real-time systems. A new cache replacement policy, called Least Critical, was proposed where data within a critical address space is given higher preference in the cache. Our design enables fine grained control over classifying task data as critical, and allows run-time configuration of a critical address space to better manage cache performance. Our experimental results show that the cache miss rate of a critical task is reduced by up to 70% when using LC cache in comparison with LRU cache. We also show that increasing critical data size deteriorates the performance of non-critical tasks. In order to manage overall performance of the application, we recommend limiting critical data size to less than cache size, or switching to a LRU cache policy at run-time when this threshold is surpassed. Avenues for future work include, 1) extending the analysis to instruction cache and 2) enabling support for data with multiple criticality levels.

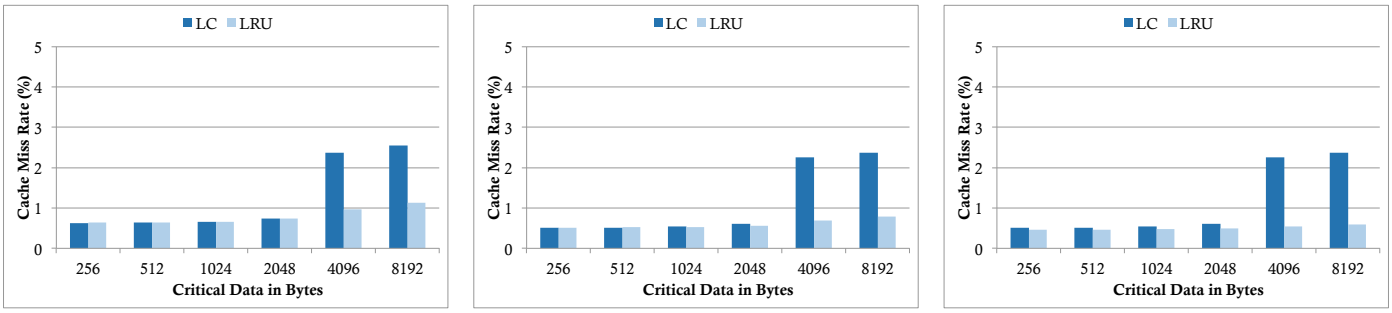
Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under award CNS-1060337, and by the Air Force Office of Scientific Research (AFOSR) under award FA9550-11-1-0343.



(a) Critical Task Period = 50 ms (b) Critical Task Period = 100 ms (c) Critical Task Period = 200 ms

Fig. 3. Critical Task: Performance of LC cache when compared to LRU cache. Critical task run with CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms



(a) Critical Task Period = 50 ms (b) Critical Task Period = 100 ms (c) Critical Task Period = 200 ms

Fig. 4. Overall Application: Performance of LC cache when compared to LRU cache. Critical task run with CRC, FDCT, Compress, and FIR. Non-Critical Task Period = 200 ms

REFERENCES

[1] A. Asaduzzaman, F. N. Sibai, and A. Abonamah, "A dynamic way cache locking scheme to improve the predictability of power-aware embedded systems," in *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*. IEEE, 2011.

[2] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 101–110.

[3] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future." OCG, 2010, pp. 137–147.

[4] C. J. Kenna, J. L. Herman, B. C. Ward, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *Euromicro Conference on Real-Time Systems*, 2013.

[5] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical os-level cache management in multi-core real-time systems," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 80–89.

[6] J. Kim, I. Kim, and Y. I. Eom, "Code-based cache partitioning for improving hardware cache performance," in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ACM, 2012, p. 42.

[7] B. Lesage, I. Puaut, and A. Seznec, "Preti: Partitioned real-time shared cache for mixed-criticality real-time systems," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM, 2012, pp. 171–180.

[8] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Enabling software management for multicore caches with a lightweight hardware support," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 14.

[9] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 45–54.

[10] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1864–1871.

[11] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 300–303.

[12] Y. Tan and V. Mooney, "A prioritized cache for multi-tasking real-time systems," in *Proc., SASIMI*, 2003.

[13] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 1, p. 4, 2007.

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[15] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 89–102.