Software Based Cache Partitioning for Mixed-Criticality Systems

Colin Burgin, Charles Suslowicz

***Abstract*—Cache partitioning provides an avenue for better predictability in real-time system execution. Mixed-critical systems that allow for increased predictability are easier to schedule. Significant previous work exists to show the value of cache partitioning and a number of techniques to support its implementation. Unfortunately, many of these techniques require custom hardware implementations or the support of a full underlying operating system. This work demonstrates the potential to bridge the gap between these previous solutions and provide a software based cache partitioning scheme through page coloring that does not require a custom hardware implementation or a full operating system to function. This will be especially applicable in mixed-criticality contexts as the combination of hard real-time tasks (HRTT) and soft real-time tasks (SRTT) benefits from both the predictability of a partitioned cache and the increased performance of shared resources on the respective task types. A simulation evaluates our method and shows that tasks with larger partitions preform better than tasks with smaller partitioned areas. These results are in line with existing work, and demonstrate the potential for this scheme to be fully implemented for additional testing and evaluation.**

***Index Terms*—mixed-criticality, cache partitioning, page coloring, real-time systems, real-time operating system.**

I. Introduction

THE term partitioning covers a broad range of techniques for reducing the uncertainty inherent in cache utilization. Through a performance trade off from generally lower cache utilization, greater predictability can be achieved and depending on the specific implementation performance and processor utilization of the system can be improved [1].

Real-time systems must meet specific time constraints while reacting to real world stimuli [2]. This separates them from normal computing systems which must only correctly complete a task, without regard for the time required in the computation or any delays encountered during execution. The introduction of this time requirement dramatically effects the operation of real-time systems compared to regular computer systems, and creates a number of conflicts with developments from regular computer systems that increase performance at the cost of deterministic behavior. The introduction of hierarchical cache systems is an example of an improvement in regular computing systems that dramatically increases performance, but the increased uncertainty in execution time often prevents or limits their use in some real-time systems [3]

Specifically, hard real-time systems that must enforce tight timing considerations on their tasks often cannot benefit from a standard cache architecture. Soft real-time systems, which are willing to occasionally have a task miss a deadline do often benefit from the performance increases afforded by multi-level cache hierarchies. This disparity previously influenced the architecture design of hard and soft real-time systems, but did not create contention since each system was implemented separately [4], [5], [6].

With recent increases in processor capability, a number of real-time systems have begun to operate in a mixed-criticality context. A mixed-criticality context is one where a single processor is responsible for both hard real-time tasks and soft real-time tasks [7]. This adds more contention to the systems design, especially as the hard real-time tasks require a high level of deterministic behavior and he soft real-time tasks often benefit from additional performance. This conflict manifests directly in the design of the cache for a mixed-criticality system.

One method of mitigating this conflict is through cache partitioning [6]. Though cache partitioning it is possible to ensure the hard real-time tasks enjoy deterministic behavior while also allowing the potential for significant performance improvements in soft real-time tasks.

There are numerous methods of achieving cache partitioning with various issues, drawbacks, and benefits. This paper explores the potential use of a dynamic software based cache partitioning for use in mixed-criticality real-time systems. Specifically, the implementation and overhead costs of cache-related preemption overhead as well as cache partition size changes are explored for the feasibility of such a cache partitioning system in a real-time system supporting tasks of different criticality levels.

This paper uses a software simulation test bench to validate applicability of a software based cache partitioning system to mixed-criticality real-time systems. Through simulation, the worst case conditions for such a cache partitioning scheme are demonstrated, highlighting the potential for a dynamic system that could avoid such behavior. Additionally, the page coloring based cache partitioning scheme is shown to provide proper isolation, at a performance cost, without requiring the use of customized hardware or a complete operating system memory management framework. Such a system, once implemented, may prove useful in a variety of real-time systems where other cache partitioning methods, either hardware or software based, are not feasible.

Colin and Charles are with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061 USA
Email: {cburgin, cesuslow}@vt.edu
Project Due 12 December 2016

II. Background

A. Real-Time Systems

Real-time systems are designed to be predictable and meet the time requirements of each task [2]. This differs from standard computing systems which aim to be fast and thereby minimize the average execution time of each task. This key difference has led to the use of real-time systems in a large number of critical systems throughout the world including areas such as avionics, military, automotive, and industrial control. In all of these cases, the ability of the real-time system to guarantee a behavior within a certain time constraint following an environmental event or stimulus is key to the success of the system [2].

Within the tasks handled by real-time systems there are two major categories: hard real-time tasks and soft real-time tasks. Hard real-time tasks (HRTT) are tasks that cause a catastrophic failure, such as loss of life, if they are not completed within their specified time frame. An example of a HRTT would be an automobile's braking system. If a standard computing system was responsible for braking a vehicle after the command was issued, it may work normally many times, but if another system process took too long to complete or otherwise monopolized system resources the vehicles brakes would become unresponsive. This is unacceptable, and the system must be designed to ensure, and be verifiable, that the brakes will operate when requested.

The other category of real-time tasks are not as critical. Soft real-time tasks (SRTT) have timing requirements, but when they cannot be satisfied it is acceptable for the system to delay or cancel the task without significant consequences. An example of a SRTT might be a periodic temperature reading of a non-critical component. If the task is delayed, the measurement can be skipping and taken in the next period without major degradation of the system [8].

The differences between these two task categories significantly impacts design decisions of real-time systems [6]. SRTTs allow a designer greater freedom to optimize the systems performance because all tasks can be treated as equal. This translates to scheduling more tasks per processor or sharing additional resources among tasks to reduce system cost. The flexibility of SRTTs has allowed for a lot of the improvements in this design space. However, the flexibility that designers have generally been able to leverage is not present in HRTTs. Designers have to take a much more pessimistic view on shared resources between tasks because if a HRTT is not able to execute in time then the system has failed [9].

B. Mixed-Criticality Systems

Historically, by supporting HRTT and SRTT on different systems designers were able to make decisions based on the type of task being executed. Recently, advances in processor power and capability have led to situations where systems support both HRTT and SRTT on the same processor. The analysis of such systems is more complex as the different confidence levels and deadline importance increase the complexity of analysis [10]. Additionally, these mixed-criticality systems create a new design challenge where predictability and optimization concerns at as opposing forces in the design and optimization of the system [6].

Significant work has been done to improve support for a mixed criticality environment. There has been work in hardware designs specifically geared toward mixed criticality tasksets such as FlexPRET and PRETI [7], [9]. These hardware architectures provide unique features that support the conflicting requirements of performance for SRTT and predictability for HRTT. PRETI specifically supports an interesting feature in the cache's awareness of the task which has populated each cache line [9]. This is a concept this paper attempts to bring to a software based solution.

The behavior of the cache in a mixed criticality system is important because of the different requirements that HRTTs and SRTTs force onto the designer. Other works showed us that isolating the HRTTs is a solution which can provide more predictability and possibly less pessimistic worst-case execution times (WCET) than would be expected in a shared cache [5]. Additionally, some architectures support sharing of the remaining resources among less critical tasks. This can lead to improvements in their overall performance by allowing them to benefit from a larger shared cache [1].

C. Cache Partitioning

A concept that has seen a significant amount of discussion is the partitioning of a cache to support a real-time system [1], [9], [5]. Through partitioning it is possible to ensure that only certain tasks have the ability to use sections of the cache. This can be accomplished through a number of techniques such as: way-based partitioning, set-based partitioning (coloring), a combined approach, or other hardware supported techniques. By separating the cache in this way, previous work has shown that it is reasonable to reduce pessimism in WCET estimations for isolated tasks and possibly allow better use of the resources available on the system [1], [11].

A number of excellent examples of hardware supported cache partitioning are discussed by Gracioli et al. in [1] and provide excellent solutions when customized hardware fits the project. The hardware supported projects described provide interesting solutions and straight forward support for partitioning the cache. The approaches differ, but this paper will focus on PRETI as a particular hardware based solution that allows the cache to track which tasks have written into which cache lines [9]. Unfortunately, the requirement of custom hardware may limit the use of this type of cache partitioning. It cannot be easily implemented on cheaper commercial systems, except as a softcore on an FPGA, until one of these architectures becomes widely available.

Software based cache partitioning methods are also discussed by Gracioli et al. and generally fall into two categories: compiler supported partitioning and dynamic partitioning. Compiler supported cache partitioning relies on a customized compiler and linker to create an executable that is structured to always send data from a particular task to the same section of the cache [1], [11]. This is a very interesting concept, and has been demonstrated successfully by groups like Plazar et al.,

but it is limited to partitioning the cache at compile time for an executable [12]. This limits the portability of the executable, and requires knowledge of the cache architecture at compile time.

The second software based method of cache partition is dynamic partitioning. This is generally done in conjunction via page coloring and often in conjunction with a virtual memory system [13], [11]. It is not discussed as frequently for real-time systems because of the associated overhead, but it does appear to have potential is an implementation can be developed that does not introduce too much uncertainty. COLORIS is a dynamic cache partitioning system developed by Ye et al. and based upon the Linux Memory Management system [14]. COLORIS is not aimed at real-time systems, but is very interesting as it provides a clear road-map for the dynamic partitioning of the cache if its concepts can be implemented in a real-time environment.

III. Method

Our goal was to test a dynamic cache-partitioning scheme for use on a mixed-criticality real-time system. By using separate partitions for hard real-time tasks (HRTT) and the remaining soft real-time tasks (SRTT) overall system performance should be more deterministic, but this determinism and flexibility will introduce additional overhead compared to a hardware based cache-partitioning scheme, but allow its use on non-custom hardware.

The design of our dynamic cache partition scheme is based on the concepts employed in the PRETI hardware architecture, tracking cache usage by task, realized through cache coloring and a minimal virtual memory page system similar to the concepts supporting COLORIS [9], [14]. This design will allow us to ensure that data from critical tasks is not evicted from the cache by less critical tasks. The COLORIS architecture aimed at general improvements to quality of service, so many of the components and techniques highlighted by Ye et al. were not necessary to provide the cache partitioning necessary for this design.

As a virtual memory based scheme, all tasks operate within their own virtual memory space. This prevents the need for the executable to be compiled with previous knowledge of the cache geometry or the tasks priority. Once created, the task is assigned a page table that handles mapping the process's virtual memory space to its allowed cache partition via a series of colors. This accomplished a similar role to the *free coloring* described by Liedtke et al. but without having to sacrifice a portion of the available physical memory space or requiring additional hardware to remove the color bits from the address [13].

The specific operation of our scheme was to utilize a task map that indicated which cache sets were available to each process. During execution, the virtual memory pages are mapped to physical memory in areas matching the tasks assigned colors. This effectively partitions the cache and prevents tasks with exclusive colors from interfering with the cache contents of each other. Even if a task is preempted, if no other tasks are assigned its cache color, the process should see no Cache Related Preemption Delay (CRPD) upon resumption of execution. CRPD will still occur if tasks share a cache color, as would be expected for SRTT where predictable execution is less important.

By structuring the cache management system in this way, it is possible to vary the number of available cache colors by changing the configure page size. This allows a high level of customization in the size of the resulting cache partitions, and could potentially also allow re-partitioning of the cache during after processes have begun execution. Re-partitioning the cache in this way would not be acceptable for HRTTs which require very predictable execution, but would prove useful on a system where the critical tasks only execute at certain intervals, and making the full cache available at other times would increase performance in SRTT tasks.

This system is significantly less robust than the COLORIS architecture described by Ye et al., but should not require a fully functioning operating system to support its management. By stripping the concept down to the minimal ideas of page color, like those described by Liedtke et al. we hope to allow the flexibility available in a software supported page coloring based cache partitioning system to work in a constrained real-time system environment [13], [14].

IV. Implementation

In order to have full control over the design we had to implement all of the different hardware and software components discussed above. Our simulator has four main components; a trace generator, a trace translation unit, a page table, and a cache. Additionally, to interface with our simulator we created a top-level simulation unit which is used to run our design simulation. The simulator accepts a number of different parameters to including: cache size, block size, page size, physical memory size, trace algorithm, and the partitioning style, etc. Our design builds off of the ideas from [9], [6] except the cache management is handled completely in software. This allows our implementation to run on any real-time system because it is architecture independent.

Our cache implementation mimics a standard LRU cache that could be found on any real-time embedded system. Using our knowledge of computer architecture as well as [1] and a reference we constructed a cache that supports any cache size, block size, or associativity/mapping. After building the cache based on the provided parameters, it is passed a trace file which contains a list of read/write operations and their associated address. The cache operates on that specific operation and returns whether the operation was a hit or miss. As stated previously, it is important that the cache has no idea what task is writing to it, the criticality of that task, or the partition or color of the task. All of that is handled in software so that the our partitioning scheme can work correctly on an device.

Before an instruction reaches the cache there are a few things that must be handled. Our page table exists as an intermediary between the RTOS and the cache. Tasks are allotted their own page table which is used to provide a mapping from the virtual space to the physical space. The page

table tracks which task each operation is associated with, its virtual address, and its partition. It uses this information to assign the operations to a color and provide them with their physical address.

In order to properly test our design we simulated read/write operations to memory. To do this we created a number of different trace generation algorithms, but ultimately limited our tests to two types: random access and worst case. The random access traces attempted to simulate realistic program operation by generating groups of sequential memory accesses at random locations in the processes virtual memory space. This method attempted to mimic the locality of normal program processes, while still subjecting the system to a large variety of memory access locations. To do this, we choose a random location in the task's allotted physical memory, and created up to 32 different operations sequentially in the same space. This simulates locality that you could find in a normal programs execution.

When generating purely random read/write operations we found very little to no improvement in a partitioned cache design. We already knew that "cache partitioning is an effective technique for achieve(ing) performance isolation among tasks" [15]. However, we wanted to show that in specific circumstances a partitioned cache where high-critical tasks are given more space actually improves the performance for those tasks and doesn't just maintain equal performance. In order to achieve this we implemented an algorithm that would walk through the cache in such a way that would cause tasks with smaller partitions to always have misses while tasks with larger partitions would have many hits. A larger partition has more sets available to it. This is exploited by having as many read/write operations as the largest partition has sets. When traces are generated for the smaller partitions we observe a lot of misses because there are not enough sets to hold the data, so the cache is constantly evicting data while the larger partition is able to store all of the data.

TABLE I
PERFORMANCE OF FULLY SHARED CACHE WITH 8 TASKS

Memory Size: 2M		**Cache Size**: 128K	
Page Size: 4K		**Block Size**: 32	
Task ID	**Hit Rate**	**Allowed Cache Sets**	**Colors**
0	0.7361	0-4095	0-31
1	0.7340	0-4095	0-31
2	0.7220	0-4095	0-31
3	0.7277	0-4095	0-31
4	0.7312	0-4095	0-31
5	0.7309	0-4095	0-31
6	0.7275	0-4095	0-31
7	0.7423	0-4095	0-31

TABLE II
PERFORMANCE OF A PARTITIONED CACHE

Memory Size: 2M		**Cache Size**: 128K	
Page Size: 4K		**Block Size**: 32	
Task ID	**Hit Rate**	**Allowed Cache Sets**	**Colors**
0	0.6952	0-3071	0-23
1	0.0310	3072-3199	24
2	0.0399	3200-3327	25
3	0.0383	3328-3455	26
4	0.0178	3456-3583	27
5	0.0186	3584-3711	28
6	0.0450	3712-3839	29
7	0.0220	3840-3967	30
8	0.0403	3968-4095	31

V. RESULTS AND EVALUATION

The software implementation of the cache was used to simulate a number of different memory access traces for various cache size, task-sets, and configurations. Memory access traces were ultimately generated by two different algorithms to explore expected and worst case cache behavior. The results from the locality influenced randomly generated traces are shown in Table II. This depicts the increased hit/miss rates for the process that was allocated a larger cache partition by receiving a larger color assignment. All of the tasks in this example were kept on separate partitions to highlight the effect of the partitioning and demonstrate its feasibility.

Not unexpectedly, the overall performance of tasks was better when they were all assigned to share the entire cache, despite cache invalidations when the executing process switched. This is shown in Table I, and highlights the performance gains that are available for tasks that can share a larger cache resource. In fact, the hit/miss ratio for tasks in a shared cache appeared to converge closely on the estimated hit/miss ratio of a single sequential memory access created by the random trace generator. This demonstrates the need for further work to test this method of cache partitioning in a live system, with a realistic task-set.

The worst case, Table IV shows the value of a partitioned cache, and the potential scenario where none critical partitions are never able to achieve a cache hit. This was created with a synthetic memory trace described in the implementation section, but it nonetheless validated the simulation providing the expected behavior for the smaller partitions worst case of memory accesses. Additionally, it provides a potential avenue for future work to investigate detecting this pattern of behavior and changing color allocations to alleviate its occurrence.

Finally, when sufficient memory and cache space is available the system is able to allow some tasks to share cache partitions while other remain isolated, shown in Table III. In this case, one which would best benefit a mixed-criticality context, the cache does show a small improvement in performance for tasks with shared resources compared when all tasks are isolated, as shown in Table II. The shared partitions are still much smaller than the entire shared cache, as shown in Table I, so the performance is still much worse than the fully shared case. The improvement, though small, over a fully partitioned cache does support the use of this type of cache partitioning in a mixed-criticality context and potential applications if implemented fully.

TABLE III
PERFORMANCE OF A SHARED PARTITIONED CACHE

Memory Size: 2M		**Cache Size**: 64K	
Page Size: 4K		**Block Size**: 32	
Task ID	**Hit Rate**	**Allowed Cache Sets**	**Colors**
0	0.3439	0-767	0-5
1	0.3459	768-1535	6-11
2	0.0859	1536-1663	12
3	0.0591	1536-1663	12
4	0.0717	1664-1791	13
5	0.0751	1664-1791	13
6	0.0558	1792-1919	14
7	0.0822	1792-1919	14
8	0.0831	1920-2047	15
8	0.0681	1920-2047	15

TABLE IV
WORST CASE PERFORMANCE DEMONSTRATION

Memory Size: 1M		**Cache Size**: 128K	
Page Size: 4K		**Block Size**: 32	
Task ID	**Hit Rate**	**Allowed Cache Sets**	**Colors**
0	0.9500	0-1023	0-7
1	0.9500	1024-2047	8-15
2	0.0000	2048-2559	16-19
3	0.0000	2560-3071	20-23
4	0.0000	3072-3583	24-27
5	0.0000	3584-4095	28-31

VI. FUTURE WORK

Future efforts include the realization of the algorithms created for this simulation in a more realistic environment. Research into a potential real-time operating system (RTOS) based virtual memory manager that would be compatible with the page coloring system developed here would be the next step. Following such an implementation, improving performance by moving away from Python and providing more realistic task behavior, it would be possible to validate the performance of an actual mixed-criticality task-set and the accompanying system behavior. Additionally, such a system would be useful for schedulability analysis of various task sets with and without partitioned caches.

VII. CONCLUSION

This project demonstrated the feasibility of a software based cache partitioning scheme for mixed-criticality real-time systems. The simulation environment showed that a software based partitioning scheme is viable and can ensure the performance of tasks allocated larger exclusive cache partitions compared to other tasks. Additional work in the area to implement a virtual memory management framework compatible with the supported real-time systems would allow the scheme outlined in this paper to support mixed-criticality task-sets on standard hardware through cache partitioning. Increasing the predictability of HRTT performance, and allowing the use of less pessimistic WCETs during schedulability analysis.

APPENDIX
SOURCE CODE

The source code created to support the partitioned cache simulated is included with this paper as a zipped directory. It consists of a number of Python 3 scripts and an examples file described below. All testing was done using Python 3.5.2 on an Ubuntu 16.04 system. The purpose of each file is outlined below:

`examples.txt` - This file contains the specific flags used in the tests highlighted in the results section. These commands can be used to replicate the test, or used as a starting point for similar tests with different cache geometries.

`cache.py` - This file instantiates the `cache` class used for the simulation. The `translator.py` file actually creates an instance of the cache used for all testing.

`pagetable.py` - This file instantiates the `PageTable` class used for virtual to physical memory mappings. Like the `cache` class, the actual page table objects are created in the `translator.py` script.

`trace_generator.py` - this file contains the functions used to create memory traces. It contains a number of attempted trace generation functions, the two actually used for worthwhile results were `build_set_evil_element_list` for worst case scenarios and `build_virt_set_element_list` for all other cases.

`simulate.py` - This is the script used to start the simulation. It parses command line arguments, formats some of the generated output, and handles the separate calls to `trace_generator.py` and `simulate.py`.

`translator.py` - This file conducts the actual simulation of a trace through the use of a `PageTable` and `cache` object. The trace data is parsed, tasks are mapped to appropriate cache sets, a cache is created based on the command, a page table is created for each task, and then the elements of the trace are evaluated against the page table for translation to physical addresses then against the cache to be either a hit or miss. Finally, results and statistics are collected for reporting.

`traces` - directory where a copy of any created traces are stored

`results` - directory where a copy of any created results are stored

ACKNOWLEDGMENT

The authors would like to Dr. Haibo Zeng, Virginia Tech, for the topic to pursue and Zaid Al-Bayati, McGill University, for additional guidance, reference suggestions, and suggestions.

REFERENCES

[1] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 32:1–32:36, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2830555

[2] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, 3rd ed. New York: Springer, 2011.

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

[4] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 34–43.

[5] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson, "Cache sharing and isolation tradeoffs in multicore mixed-criticality systems," in *Real-Time Systems Symposium, 2015 IEEE*, Dec 2015, pp. 305–316.

[6] N. G. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones, "Cache design for mixed criticality real-time systems," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct 2014, pp. 513–516.

[7] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "Flexpret: A processor platform for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 101–110.

[8] D. B. Kirk and J. K. Strosnider, "Smart (strategic memory allocation for real-time) cache design using the mips r3000," in *[1990] Proceedings 11th Real-Time Systems Symposium*, Dec 1990, pp. 322–330.

[9] 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*, ser. RTNS '12. New York, NY, USA: ACM, 2012, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/2392987.2393009

[10] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Dec 2007, pp. 239–243.

[11] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "On the effectiveness of cache partitioning in hard real-time systems," *Real-Time Systems*, vol. 52, no. 5, pp. 598–643, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11241-015-9246-8

[12] S. Plazar, P. Lokuciejewski, and P. Marwedel, "WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems," in *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, ser. OpenAccess Series in Informatics (OASIcs), N. Holsti, Ed., vol. 10. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 1–11, also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2009/2286

[13] J. Liedtke, H. Hartig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, Jun 1997, pp. 213–224.

[14] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: A dynamic cache partitioning system using page coloring," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 381–392. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628104

[15] C. Wang, Z. Gu, and H. Zeng, "Integration of cache partitioning and preemption threshold scheduling to improve schedulability of hard real-time systems," in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015, pp. 69–79.