Experimental Platform for Testing Cache Allocation Policies to Improve Linux Real-Time Behaviour

Aléxis Génèrès, Michaël Lauer

Experimental Platform for Testing Cache Allocation Policies to Improve Linux Real-Time Behaviour

Aléxis Génèrès[1]
[1] CNRS LAAS, 7 avenue du colonel Roche
F-31400, Toulouse, France
Email: ageneres@laas.fr

Michaël Lauer[1,2],
[1]CNRS LAAS, 7 avenue du colonel Roche
[2]Univ de Toulouse, UPS
F-31400, Toulouse, France

***Abstract*—Automotive embedded systems have an increasing need for computing power. To address this issue, we can implement critical and non-critical tasks in the same multicore processor. A disadvantage of this kind of processor is the indeterminism it involves due to its complexity. New technologies, like dynamic allocation of cache memory, allow reducing the impact of this indeterminism. In this article, we provide an experimental approach to verify if the dynamic allocation of the cache memory of Intel (CAT Intel) is efficient.**

***Index Terms*—multi-core, real time, multiple criticity, cache allocation**

I. Introduction

The growing need for computing power in embedded systems is leading companies to use multi-core processors, in particular in the automotive domain for both economic and technical reasons. Open source operating system is an other attractive opportunity, namely Linux. The increased number of onboard applications requires combining high criticality tasks with low criticality tasks on the same multi-core processor. Unfortunately, the uses of multi-core processors induce a lack of predictability of their temporal behavior due to their shared resources: memories, communication buses... These factors can provoke deadlines violation for high criticality tasks, and also a sub-optimal use of the computing power for low criticality tasks [1].

In this paper, we target one factor: the cache memory shared between the cores of a processor. Our goal is to evaluate the gain of a dynamic cache allocation mechanism regarding deadlines respect and processing power performance. Furthermore, we want to know if the use of these mechanisms can be implemented with a minimum effect on the background tasks. On the one hand, a static allocation allows to isolate the high criticality tasks and thus, to limit the interferences linked to shared resources. However, on the other hand, the cache memory, in case of a static allocation, is then no longer fully available for other tasks, even if the high criticality tasks are terminated. This limits the use of the processor. In this study, we will therefore focus on dynamic allocation to avoid this behavior.

In this preliminary work, we will realize a set of experiments using high criticality tasks and low criticality tasks on a single multi-core processor. More specifically, we use the technology of cache memory allocation to isolate the cache memories of the tasks having a different level of priority. The goal is to evaluate the impact of dynamic low-level cache allocation mechanisms on the side-effect of shared resources.

To achieve this evaluation, we will use the *Earliest Deadline First (EDF)* scheduling of Linux. Based on the works of Lelli [2], we can use the EDF scheduling of Linux and guarantee the real-time requirements if we limit the computing power at 90% on one core. For our experiments, we will also use CAT technology of Intel. In section II of this article we describe this work. In this section, we also define our experimental approach and tools to analyze the last level of cache memory mechanism. In section III, we detail our experimental platform and our protocol to evaluate the mechanism.

II. Experimental approach and tools

A. Experimental approach

The goal behind our experimental protocol is to improve the execution determinism of the high criticality task while also ensuring optimal performances for the low criticality tasks. To this end, we evaluate the relevance of the dynamic allocation of cache memory on the shared cache memory.
To do this, we simulate one high criticality task with 12 low criticality tasks. After that, we will measure the response time of the high criticality task to evaluate the non-violation of the deadline and thus the execution determinism. In a second time, we estimate the computing power that we can use for low criticality tasks by measuring the IPCs of the 12 low criticality tasks.
Therefore, the experimental approach to evaluate the impact of dynamic low-level cache allocation mechanisms for reducing the effect of shared resources is composed of three phases.

1) Without cache memory allocation.
2) With static cache memory allocation.
3) With dynamic cache memory allocation.

If we do not observe any ”deadline miss” nor any behavior side-effect with dynamic allocation, we can state that dynamic allocation has no negative impact on the high criticality task.

B. Tools

In our experimental framework, the high criticality task is scheduled with the EDF scheduling policy of Linux (named *SCHED_DEADLINE*). This scheduler is characterised by three key parameters: *”runtime”*, *”deadline”* and *”period”*. The ”runtime” parameter represents the budget allocated by the OS to perform the task before its deadline. This budget is refreshed

at each new period. If this budget is fully consumed, the task is suspended and we consider its deadline as missed. We manage the low criticality tasks using the default scheduling policy of Linux: "time-sharing".

For cache memory allocation, we use the Intel CAT (Cache Allocation Technology)[3]. CAT allows us to allocate a subspace of memory of the last level cache of the processor. We can use CAT for multiple cores or processes. The allocation is done via a software lock: a mask that changes the write permissions only. It's granularity is determined by the number of *ways* in the CPU. For example, in a 8 *ways* last level of cache, we can allocate from 1 to 8 *ways* (zero is not possible) to one or multiple processes. The maximum number of allocation depends on the CPU models.

Finally, we use BCC (BPF Compiler Collection)[4] to collect data (IPC and response times). BCC permit our observation mechanisms and measures to be implemented. BCC is a group of elementary tools which allows us to program and use BPF (Berkeley Packet Filter) probes in order to collect the necessary data for our experiments. BCC introduces a low overhead in terms of computing power and provides key elements to implement fine-grained instrumentation.

III. Experimental protocol

The overall approach to address the problem is composed of three phases. Each phase consists of the execution of the high criticality task alone, then, in a second step, the execution of the high criticality task and low criticality tasks simultaneously. This step aims to generate interference. The three phases are the following:

- Without cache memory allocation.
- With static cache memory allocation.
- With dynamic cache memory allocation.

For the realization of this protocol, the experimental platform uses two processors for the time being. One is an Intel Xeon Bronze 3204 (named Xeon 3204). A Xeon 3204 has 6 cores without hyperthreading, its last level is an 11-way set-associative cache with a size of 8.25MB. The L2 caches are of a size of 1MB each and are not shared by multiple cores. The second is an Intel Celeron J3455. A J3455 has 4 cores without hyperthreading, its last level is an 8-way set-associative cache with a size of 2×1MB.
In the remainder in this paper, we use the Xeon 3204 as the main processor.

Based on the information about the last level cache of our processor, we can explain the use of CAT in this experimentation. Using CAT we can exploit two times 11 *ways* for the allocation. Each *way* is an allocation of 0.75 MB of memory space. Since our high criticality task needs 3 MB of cache memory space, to isolate it, we will use CAT to allocate a sufficient number of ways : 5 ways. We will also deactivate the write authorization in the same memory space for the low criticality tasks.

The operating system we will work on is a 5.12.5 Kernel. It allows us to have access to the last version of BCC. We will use BCC to save the start date of the high criticality task. We will also need, using this compiler collection, to save any violation of the deadline for the high criticality task and, to measure the Instruction per Cycle (IPC) of each core.

A. Step by step protocol

Each of these phases (without allocation, with a static allocation, and with a dynamic allocation) regroup two steps; the first with the high criticality task alone and the second with the high criticality task and the low criticality tasks in parallel.
As we use the EDF Linux schedule for the high criticality task, we must specify the three parameters of this scheduler: runtime, deadline, and period.

For our experiment, the runtime is 150ms and the period is equal to the deadline: 200 ms. Both of the tasks (high criticality and low criticality) use an entry data size of 3 MB. Indeed, to provoke the effect of the shared memory, the entry data size needs to be greater than the size of L2 cache memory size which is 1MB for the Intel Xeon 3204.

1) Without cache memory allocation phase: as mentioned before, the first step is to launch the high criticality task alone with the observation tools. This step intended to collect the reference response times.
The second step is to launch the high criticality task and the low criticality tasks at the same time. The low criticality tasks are represented by 12 stress tasks, one per core, which will flood the cache memory. Thereby, we could measure the effect of noisy neighbors on our system. This effect is due to the interferences of the memories shared by other tasks.

2) With static cache memory allocation phase: the static cache memory allocation will be effective until the end of this step. To set the allocation, we allocate to high criticality task five successive *ways* among the 11 available in the processor. We can do this allocation using the CAT intel technology.

This allocation will be exclusive to the high criticality task. The allocated memory size is 3.75 MB (0.75MB by *way*). In fact, it needs to be greater than 3 MB which is the size of the data used by the high criticality task.

The first step, which is to launch the high criticality task alone, allow us to observe the effect of cache allocation on the high criticality task response time. This effect should be insignificant.

The second step, which is to load the system with our 6 stress tasks, should give us the same results, i.e no modification of our response times because the allocation protects the high criticality task. Since we use only five of the 11 ways to the high criticality task, we suppose that the absence of the six other *ways* will impact the number of instructions per cycle for the low criticality tasks.

To observe this result we compare:

- the average response time.

- the average IPCs of all the cores.

3) With dynamic cache allocation phase: for this phase, we also use five of the existing *ways* for the allocation. We will deactivate this allocation when the high criticality task has terminated and reactivate it at the beginning of a new period.

To analyze the impact of the dynamic allocation, we compare the dynamic and the static allocation regarding the average response.

To analyse the computing power of the low criticality tasks, we compare the dynamic and the static allocation regarding the IPCs. This comparison is done while the stress tasks are activated. We should observe a higher IPC in the case of dynamic allocation. This would conclude about the efficiency of this mechanism towards low criticality tasks.

IV. Related works

In this section, we will detail some solutions to make coexist a high criticality task and tasks of low-level criticality on the same multi-core processor.

The first solution is illustrated by the works of Suzuki and others [5]. It suggests allocating for each core a subspace of the cache memory. Then, to set the core affinity of the high criticality task to only one core and assign all the other cores to the low criticality tasks. This solution has been proven to limit the impact of the shared resources. However, it is quite pessimistic due to the proprietary of one core, which can be not fully used, to the high criticality task. This solution showed similarities with the static allocation of cache memory (risk of over-allocation).

Another type of solution, that we call "all-or-nothing" is to execute the high criticality task while we deactivate all the low criticality tasks. Such is the case of the works of Kritikakou and others. [6] which shows how to guarantee the real-time constraint with a two-step method. First, by executing tasks with different levels of criticality in parallel. Then, by deactivating the low criticality tasks based on a computation of the RWCET (remaining worst-case execution time). Other works of Kiritikakou and others. [7] and Girbal and others. [8] uses this same method. In our experimental approach, we want to be less pessimistic and keep the low criticality tasks activated.

Finally, a solution proposed by Xu and others. [9] uses CAT technology to allocate cache memory on Virtual Machines. The number of Virtual Machines is equal to the different levels of criticality. Then, tasks are executed on these Virtual Machines depending on the level of criticality. The disadvantage of this method is that the low criticality tasks will never use 100 % of cache memory due to pre-reservation.

V. Technical details

To use the experimental platform the high criticality task need to to be schedule with Sched deadline. We use the sched_yield command to end the job and release the scheduler.

We create a python script using BCC to measure the response time of the high criticality task. We trace the deadline scheduler of Linux. More precisely we execute a C script at any update of the deadline scheduler at the kernel level. This C script aims to save the state of the high criticality task at the moment of the update. If the high criticality task is not yielded and not throttled, then the task is currently running. If the high criticality task is yielded then it's the end of the measure of the response time. And if the task throttled (consume all the runtime budget) and it is not yielded then it is considered has a deadline miss. At the end of this C script, the data are push in the user level (and come to the python script) and the save in the text file don't impact the high criticality task.

For the low criticality task we measure the IPC of each core. Howerver to instrument the instruction, we need to place a probe. One solution is to place a fixe probe every x instructions (or cycle). But with this solution we cannot have temporal coherence during the measure. The second solution, that we use, is to create C scipt at user level which is basically a clock that use Linux Signal and don't use CPU in background. We run on each core this clock script. After, we trace the tracepoint signal generate. The C script at the kernel level (with BCC) test if the signal generated is the Signal 14 (SIG ALARM). Then, we open for each CPU, the perf event for the cycles and the perf event for the instructions. At the end we push the data in the user level (python script) with the name of application which generate the signal, the core, the instructions, and cycles. We can't calculate IPC directly due to the missing of floating type in kernel level.

VI. Conclusion

Our experimental protocol and the corresponding platform aim at analyzing the dynamic allocation mechanism of the last lever cache, on an experimental computer with an Intel processor and using Linux. Our approach is meant to estimate the impact of noisy neighbours effect on a high criticality task. Furthermore, we aim to find a compromise between the temporal requirement for the high criticality task, and as much as possible computing power to low criticality tasks.

The dynamic cache allocation uses CAT technology to allocate enough cache memory only during the runtime of the high criticality task. It is worth noting that CAT has limited precision when allocating processor ways last level cache to tasks.

If the dynamic allocation is efficient, then we will circumvent this issue by affecting different allocations to tasks during our experiments. More specifically, we allocate the sufficient number of ways for a task during its time execution only and then reallocate the same ways to another task when the first one is over.

The technology used for scheduling, which is Linux EDF, excludes the possibility to manage the core affinity for specific tasks. Regarding this second limitation, Linux is going through a process of adding real time assets on its new future Kernel. We therefore expect this limitation to be resolved in a near future.

REFERENCES

[1] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," 05 2010, pp. 36–42. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.4533&rep=rep1&type=pdf

[2] A. L. Lelli Juri, Scordino Claudio and F. Dario, "Deadline scheduling in the linux kernel," *Software practice and experience*, vol. 46, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.2335

[3] K. Nguyen, " cat cache allocation technology " 2016. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html

[4] "Bpf compiler collection (bcc)," 2021. [Online]. Available: https://github.com/iovisor/bcc

[5] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, 2013, pp. 685–692.

[6] A. Kritikakou, T. Marty, and M. Roy, "DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems," *ACM Tran. on Design Automation of Electronic Systems*, vol. 23, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3149546.3110222

[7] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 139–148. [Online]. Available: https://doi.org/10.1145/2659787.2659799

[8] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core cots," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015, pp. 8D4–1 to 8D4–15.

[9] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 211–222.