



HAL
open science

Experimenting with Dynamic Cache Allocation to Improve Linux Real-Time Behaviour

Aléxis Génèrès, Michaël Lauer, Jean-Charles Fabre

► **To cite this version:**

Aléxis Génèrès, Michaël Lauer, Jean-Charles Fabre. Experimenting with Dynamic Cache Allocation to Improve Linux Real-Time Behaviour. 11th European Congress of Embedded Real Time Software and Systems, Jun 2022, Toulouse, France. hal-04044944

HAL Id: hal-04044944

<https://hal.science/hal-04044944>

Submitted on 24 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Experimenting with Dynamic Cache Allocation to Improve Linux Real-Time Behaviour

Al  xis G  n  r  s[†],

[†] LAAS-CNRS

31400, Toulouse, France

Email: ageneres@laas.fr

Micha  l Lauer[†], Jean-Charles Fabre[†]

[†] LAAS-CNRS

31400, Toulouse, France

Email: firstName.lastName@laas.fr

Abstract—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 criticality, cache allocation

I. INTRODUCTION

The growing need for computing power in embedded systems is leading companies to use multi-core processors. 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 our mechanism of dynamic cache allocation 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 temporal tasks and thus, to limit the interferences linked to the 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 temporal 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 will 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 will evaluate the relevance of the dynamic allocation of cache memory on the shared cache memory. To do this, we will 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 will 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 will be 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 the experimental protocol, we will use a two processors system. Each processor 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.

Based on the information about the last level cache of our processors, 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 (two processors with 6 cores), 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 will allocate 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, will 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 12 stress tasks, should give us the same results, i.e we will not have any modification of our response times because the allocation protects the high criticality task. Since we use only five or the 11 *ways*, we suppose that the absence of the six other *ways* will impact the number of instructions per cycle.

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 will compare the dynamic and the static allocation regarding the

average response.

To study the computing power of the low criticality tasks, we will compare the dynamic and the static allocation regarding the IPCs. This comparison will be 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 Kritikakou 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. CONCLUSION

Our experimental protocol aims to analyze the dynamic allocation mechanism of the last level cache, with an experimental computer of two intel processors, using Linux. This experience allows us to decide on the efficiency of this kind of mechanism using a real-time task with a period of around 100ms. Furthermore, our approach is meant to limit the impact of the dynamic allocation on the low criticality task regarding the computing power. At the moment, we are developing dynamic allocation tools for our experimental platform.

CAT has a limited precision when it comes to ensuring the allocation of ways of the processor's last level cache.

The technology used for scheduling, which is Linux EDF, exclude the possibility to manage the core affinity for specific task.

About the first limitation due to CAT technology. If the dynamic allocation is efficient, then we will circumvent this issue by affecting different allocations over the time period of our experimentation. More specifically, we will allocate the sufficient ways for a task during its time execution only and then reallocate the same ways to another task when the first one is over. Thank's to that, we can artificially increase the size of the last level cache available for the tasks.

Regarding the second limitation, Linux is going through a process of adding real time assets on it's new Kernel. We therefore expect this limitation to be resolved in a near future.

After obtaining the experimental results, our outlooks are to add other high criticality tasks in our system as a first step. Then, to use more than one core for the high criticality tasks as a second step.

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.